



P.B.5818 - Patentlaan 2
2280 HV Rijswijk (ZH)
☎ (070) 3 40 20 40
FAX (070) 3 40 30 16

Europäisches
Patentamt

European
Patent Office

Office européen
des brevets

Generaldirektion 1

Directorate General 1

Direction générale 1

Clearspeed Technology PLC
3110 Great Western Court, Hunts Ground Road
Stoke Gifford, Bristol BS34 8HP
GRANDE BRETAGNE



EPO Customer Services

Tel.: +31 (0)70 340 45 00

Date

17-07-2006

Reference	Application No./Patent No. 04768907.0 - 2211 PCT/GB2004004377
Applicant/Proprietor Clearspeed Technology PLC	

Noting of loss of rights pursuant to Rule 69(1) EPC

The European patent application cited above is deemed to be withdrawn (Rule 108(1) EPC) for the following reason(s):

- a) ☐ translation of the international application into one of the EPO's official languages (Art. 158(2) EPC) not filed within the period specified in Rule 107(1)(a) EPC
- b) ☒ national basic fee
☐ search fee
☒ designation fee
☒ examination fee and/or written request for examination
not validly paid / not made within the time limit specified in Rule 107(1)(c)-(f) EPC
- (c) ☐ payment of the above fees on, after expiry of the period for payment (on 15.05.06).

MEANS OF REDRESS:

1. The loss of rights [(a)(b)] shall be deemed not to have occurred if, within a (non-extendable) period of **TWO MONTHS** of notification of this communication, the relevant requirement(s) has (have) been fulfilled and the appropriate surcharge(s) under Article 2(3b)(3c) RFees have been paid (Rule 108(3) EPC).
If fees were paid late [(c)], the requirement(s) as specified in Article 8(3)(4) RFees is (are) to be fulfilled within the same time limit.
2. If, however, the applicant considers that this finding is inaccurate, he may apply in writing for an EPO decision on the matter (Rule 69(2) EPC) within the same time limit, i.e. that specified in (1). The finding will be set aside only if it does not actually correspond to the factual or legal situation.
The applicant's rights with regard to fee payment or filing the written request for examination cannot be re-established under Article 122 EPC.
3. If, in spite of all due care required by the circumstances having been taken, the applicant was unable to **observe the time limit for filing the translation**, he will, upon application, have his rights re-established provided that the time limits and formal requirements laid down in Article 122 EPC are complied with.



Date

Sheet 2

Application No. 04768907.0

NOTE:

Applicants not having either a residence or principal place of business within the territory of one of the EPC contracting states must have the above request under Article 122 EPC or Rule 69(2) EPC filed by a professional representative authorised to act before the EPO.

RECEIVING SECTION

Enclosure: Schedule of Fees



Gebühren für in die europäische Phase eintretende internationale Patentanmeldungen in EURO geltend ab 01.04.06 *
Fees for international applications entering the European phase in EURO effective as from 01.04.06 *
Taxes pour les demandes internationales entrant dans la phase européenne en EURO applicables à compter du 01.04.06 *

Nationale Grundgebühr / National basic fee / Taxe nationale de base: Das Formblatt für den Eintritt in die europäische Phase (EPA Form 1200) wird auf Papier eingereicht: The form for entry into the European phase (EPO Form 1200) is filed on paper: Le formulaire d'entrée dans la phase européenne (formulaire OEB 1200) est déposé sur papier: Das Formblatt für den Eintritt in die europäische Phase (EPA Form 1200) wird online eingereicht: The form for entry into the European phase (EPO Form 1200) is filed online: Le formulaire d'entrée dans la phase européenne (formulaire OEB 1200) est déposé en ligne:	170,00 (160,00) *** 95,00 (90,00) ***
Recherchegebühr / Search fee / Taxe de recherche ** Anwendbar für ab dem 1. Juli 2005 eingereichte internationale Patentanmeldungen: Applicable to international applications, filed on or after 1 July 2005: Applicable aux demandes internationales déposées à compter du 1er juillet 2005: Anwendbar für vor dem 1. Juli 2005 eingereichte internationale Patentanmeldungen: Applicable to international applications filed before 1 July 2005: Applicable aux demandes internationales déposées avant le 1er juillet 2005:	1000,00 (960,00) *** 720,00 (690,00) ***
Benennungsgebühr für jeden benannten Vertragsstaat (siebenfacher Betrag = alle Vertragsstaaten): Designation fee for each contracting state designated (seven times the amount = all contracting states): Taxe de désignation pour chaque Etat contractant désigné (sept fois cette taxe = tous les Etats contractants):	80,00 (75,00) ***
Zuschlagsgebühr für die verspätete Einreichung der Übersetzung der internationalen Anmeldung oder die verspätete Stellung des Prüfungsantrags oder die verspätete Entrichtung der nationalen Grundgebühr, der Recherchegebühr und/oder der Benennungsgebühren (Regel 108 (3) und (4): 50% der betreffenden Gebühren, jedoch mindestens 520 EUR bei verspäteter Einreichung der Übersetzung und insgesamt höchstens: Surcharge for late filing of either the translation of the international application or the request for examination, or for late payment of the national basic fee, the search fee and/or the designation fee (Rule 108(3) and (4): 50% of the relevant fees, but at least EUR 520 for late filing of the translation, up to a maximum of: Surtaxe pour production tardive de la traduction de la demande internationale, pour présentation tardive de la requête en examen ou pour retard de paiement de la taxe nationale de base, de la taxe de recherche et/ou des taxes de désignation (règle 108 (3) et (4): 50% des taxes concernées, sans que le montant puisse être inférieur à 520 EUR en cas de production tardive de la traduction et sans que le montant total puisse dépasser:	1820,00 (1750,00) ***
Anspruchsgebühr für den elften und jeden weiteren Anspruch: Claims fee for the eleventh and each subsequent claim: Taxe pour chaque revendication à partir de la onzième:	45,00 (40,00) ***
Pauschalgebühr für eine zusätzliche Abschrift der im Europäischen Recherchenbericht aufgeführten Schriften: Flat-rate fee for an additional copy of the documents cited in the European search report: Taxe forfaitaire pour une copie supplémentaire des documents cités dans le rapport de recherche européenne:	25,00 (20,00)***
Prüfungsgebühr / Examination fee / Taxe d'examen Anwendbar für vor dem 1. Juli 2005 eingereichte internationale Patentanmeldungen: Applicable for international applications filed before 1 July 2005: Applicable aux demandes internationales déposées avant le 1 juillet 2005: Anwendbar für ab dem 1. Juli 2005 eingereichte internationale Patentanmeldungen: Applicable to international applications filed on or after 1 July 2005: Applicable aux demandes internationales déposées à compter du 1er juillet 2005: Anwendbar für ab dem 1. Juli 2005 eingereichte internationale Patentanmeldungen, für die kein ergänzender europäischer Recherchenbericht erstellt wird (Art. 157(3)(a)): Applicable for international applications filed on or after 1 July 2005 for which no supplementary search report is drawn up (Art. 157(3)(a)): Applicable aux demandes internationales qui sont déposées à compter du 1er juillet 2005 pour lesquelles il n'est pas établi de rapport complémentaire de recherche européenne (art. 157(3)(a)):	1490,00 (1430,00) *** 1335,00 (1280,00) *** 1490,00 (1430,00) ***

* = Siehe Beschluss des Verwaltungsrats vom 15.12.05 (ABI 1/2006, 8)
See Decision of the Administrative Council of 15.12.05 (OJ 1/2006, 8)
Voir Décision de Conseil d'administration du 15.12.05 (JO 1/2006, 8)

** = Siehe ABI 3/2006
See OJ 3/2006
Voir JO 3/2006

*** = Anwendbar für Zahlungen eingegangen bis 31.03.06
Applicable for payments received up to 31.03.06
Applicable aux paiements reçus jusqu'au 31.03.06

* = Postscheckkonto
Giro account
Compte de chèque postal

Bank- und Postscheckkonten der Europäischen Patentorganisation in EURO
Bank and Giro accounts of the European Patent Organisation in EURO
Comptes bancaires et chèques postaux de l'Organisation européenne des brevets en EURO

Staat Country Pays	Bank- und Postscheckkonten / Bank and Giro accounts / Comptes bancaires et de chèques postaux	Staat Country Pays	Bank- und Postscheckkonten / Bank and Giro accounts / Comptes bancaires et de chèques postaux	Staat Country Pays	Bank- und Postscheckkonten / Bank and Giro accounts / Comptes bancaires et de chèques postaux
AT	No. 102-133-851/00 (BLZ 12 000) IBAN AT91 1200 0102 1338 5100 BIC BKAUATWW Bank Austria Creditanstalt AG Am Hof 2 Postfach 52000 A-1010 Wien	EE	No. 10220025988223 IBAN EE24 1010 2200 2598 8223 BIC EEUH22XX SEB Eesti Ühispank AS Tornimäe 2, Tallinn EE-15010 Tallinn	LU	No. 7-108/9134/200 IBAN LU41 0027 1089 1342 0000 BIC BILLULL DEXIA-Banque Internationale à Luxembourg 69, Route d'Esch L-2953 Luxembourg
	No. 7451.030 (BLZ 60 000)* IBAN AT96 6000 0000 0745 1030 BIC OPSKATWW Österreichische Postsparkasse AG Georg-Coch-Platz 2 A-1010 Wien	ES	No. 0182-5906-88-029-0348002 IBAN ES74 0182 5906 8802 9034 8002 BIC BBVAESMM Banco Bilbao Vizcaya Argentaria S.A. (BBVA) Calle Acalá 16, 3e Planta E-28014 Madrid		No. 6421-37 * IBAN LU97 1111 0264 2137 0000 BIC CCPLULL Administration des P. & T. 38, place de la gare Chèques postaux L-1090 Luxembourg
BE	No. 310-0449878-78 IBAN BE69 3100 4498 7878 BIC BBRUBEBB ING Belgium Marnix Business Branch 1, rue du Trône B-1000 Bruxelles	FI	No. 200118-182076 IBAN FI28 2001 1800 1820 76 BIC NDEAFIHH Nordea Bank Finland plc, Aleksanterinkatu 36, 12000 Helsinki FIN-00020 Nordea	LV	Noch nicht verfügbar Not yet available Pas encore disponible
	No. 000-1154426-29 * IBAN BE26 0001 1544 2629 BIC BPOTBEB1 Banque de la Poste W.T.C., Tour II Chaussée d'Anvers B-1100 Bruxelles		No. 800013-90405 * IBAN FI27 8000 1300 0904 05 BIC PSPBFIHH Sampo Bank plc Fabianinkatu 23, Helsinki FIN-00075 Sampo	MC	No. 30004 09179 00025422154 91 (RIB) IBAN FR76 3000 4091 7900 0254 2215 491 BIC BNPAFRPPAMC BNP - Paribas Agence Monaco Charles III Avenue de la Madone MC-98000 Monaco
BG	No. 1465104501 IBAN not implemented BIC Bulbank 7, Sveta Nedelya Sq. BG-1000 Sofia	FR	No. 30004 00567 00020020463 29 (RIB) IBAN FR76 3000 4005 6700 0200 2046 329 BIC BNPAFRPPPOP BNP - Paribas Agence: Paris Clientèle Internationale 2, Place de l'Opéra F-75002 Paris	NL	No. 51 36 38 547 IBAN NL54 ABNA 0513 6385 47 BIC ABNANL2A ABN AMRO Bank NV Kneuterdijk 1, Postbus 165 NL-2501 AP Den Haag
CH	No. 230-322 005 60 M IBAN CH49 0023 0230 3220 0560 M BIC UBSWCHZH80A UBS Bahnhofstrasse 45 CH-8021 Zürich	GB	No. 86 98 72 66 (Sorting Code 20-47-35) IBAN GB10 BARC 2047 3586 9872 66 BIC BARCGB22 Barclays Bank plc International Corporate PO Box 391 38 Hans Crescent Knightsbridge GB-London SW1X 0LZ		No. 40 12 627 * IBAN NL27 PSTB 0004 0126 27 BIC PSTBNL21 Postbank N.V. Postbus 98000 NL-6800 MA Arnhem
	No. 91-649517-3 * IBAN CH79 0900 0000 9164 9517 3 BIC POIFCHBE PostFinance Operations Center CH-6007 Luzern		No. 112002002007046 IBAN GR36 0140 1120 0200 2007 046 BIC CRBAGRAAXXX Alpha Bank Athens Tower Branch 2, Messoghion Avenue GR-115 27 Athens	PL	No. 42103015080000000504086003 IBAN PL42 1030 1508 0000 0005 0408 6003 BIC CITIPLPX Bank Handlowy w Warszawie S.A. Senatorska 16 PL-00-923 Warszawa
CY	No. 0155-41-190144-48 IBAN CY68 0020 0155 0000 0041 1901 4448 BIC BCYPCY2N Bank of Cyprus 2-4 Them. Dervi Street. P.O. Box 1472 CY-1599 Nicosia	GR	No. 11764946-00239880 IBAN HU46 1176 4946 0023 9880 0000 0000 BIC OTPVHUHB OTP Bank Rt. Kőzponti Fiók Deák Ferenc utca 7-9 HU-1052 Budapest	PT	No. 2088391145 IBAN PT50 0033 0000 0208 8391 1452 2 BIC BCOMPTPL Millennium bcp Banco Comercial Português Av. Fontes Pereira de Melo, 7 P-1050-115 Lisboa
	No. 01841280/0300 IBAN CZ52 0300 1712 8010 1700 2453 BIC CEKOCZPP Československá Obchodní Banka A.S. Na Příkopě 854/14 CZ-11520 Praha 1 - Nové Město	HU	No. 309 822 01 (Bank Code 901 490) IBAN IE10 BOFI 9014 9030 9822 01 BIC BOFIE2D Bank of Ireland Lower Baggot Street Branch P.O. Box 3131 IRL-Dublin 2	RO	Noch nicht verfügbar Not yet available Pas encore disponible
CZ	No. 3 338 800 00 (BLZ 700 800 00) IBAN DE20 7008 0000 0333 880000 BIC DRESDEFF Dresdner Bank Promenadeplatz 7 D-80273 München SWIFT Code: DRESDEFF700	IE	No. 0101-38-710440 IBAN IS77 0101 3871 0440 4312 0490 80 BIC LAISISRE Id no. EPO 431204-9080 (compulsary) National Bank of Iceland Main Branch Austurstræti 11 IS 101 Reykjavik	SE	No. 99-48857939 IBAN SE08 6000 0000 0000 4885 7939 BIC HANDSESS SHB CUBB-K3 Svenska Handelsbanken S-106 70 Stockholm
	No. 300-800 (BLZ 700 100 80) * IBAN DE59 7001 0080 0000 3008 00 BIC PBNKDEFF Deutsche Postbank AG Bayerstrasse 49 D-80138 München	IS	No. 936832 01 94 (conto estero) IBAN IT21 E030 6905 0200 0936 8320 194 BIC BCITITMM700 ABI 03069 / CAB 05020 Banca Intesa Via del Corso, 226 I-00186 Roma		No. 7 41 53-8 * IBAN SE88 9500 0099 6042 0074 1538 BIC NDEASESS Nordea Bank Sweden AB/Postgirot Drottningg 4 P.O. Box 16081 S-103 22 Stockholm
DE	No. 3001014560 IBAN DK94 3000 3001 0145 60 BIC DABADKKK Danske Bank A/S Holmens Kanal Dept. Holmens Kanal 2-12 DK-1092 København K.	IT	No. 10568277 * IBAN IT89 Q 0760111 3000 0001 0568 277 BIC BPPIITRRXXX Poste Italiane C.U.A.S. Piazza Vesuvio, 6 I-20144 Milano	SI	No. 03500-1000001709 IBAN SI56 0350 0100 0001 709 BIC SKBASIXX SKB Banka D.D. Ajdovščina 4 SI-1513 Ljubljana
	No. 1551-2138837478 * IBAN DK16 3000 2138 8374 78 BIC DABADKKK BG Bank Erhvervsafdeling City Nørre Voldgade 68 DK 1011 København K.		Noch nicht verfügbar Not yet available Pas encore disponible	SK	No. 2920480237 (Bank code 1100) IBAN SK89 1100 0000 0029 2048 0237 BIC TATRSKBX Tatra Banka A.S. Branch Banská Bystrica Dolná 2 SK-97401 Banská Bystrica
DK	No. 4214-301120-1039000 IBAN not implemented BIC Türkiye İS Bankası A.Ş. Gazi Mustafa Kemal Bulvarı 8 TR-06640 Kızılay / Ankara	LT	Noch nicht verfügbar Not yet available Pas encore disponible	TR	

PATENT COOPERATION TREATY

PCT

INTERNATIONAL PRELIMINARY REPORT ON PATENTABILITY

(Chapter I of the Patent Cooperation Treaty)

(PCT Rule 44bis)

Applicant's or agent's file reference P100390PC00/CIV	FOR FURTHER ACTION See item 4 below	
International application No. PCT/GB2004/004377	International filing date (<i>day/month/year</i>) 13 October 2004 (13.10.2004)	Priority date (<i>day/month/year</i>) 13 October 2003 (13.10.2003)
International Patent Classification (8th edition unless older edition indicated) See relevant information in Form PCT/ISA/237		
Applicant CLEARSPED TECHNOLOGY PLC		

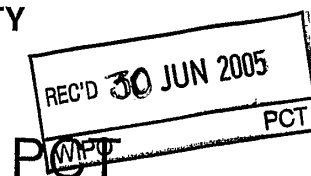
1.	This international preliminary report on patentability (Chapter I) is issued by the International Bureau on behalf of the International Searching Authority under Rule 44 <i>bis</i> .1(a).	
2.	This REPORT consists of a total of 9 sheets, including this cover sheet. In the attached sheets, any reference to the written opinion of the International Searching Authority should be read as a reference to the international preliminary report on patentability (Chapter I) instead.	
3.	This report contains indications relating to the following items: ☒ Box No. I ☐ Box No. II ☐ Box No. III ☒ Box No. IV ☒ Box No. V ☐ Box No. VI ☐ Box No. VII ☐ Box No. VIII Basis of the report Priority Non-establishment of opinion with regard to novelty, inventive step and industrial applicability Lack of unity of invention Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement Certain documents cited Certain defects in the international application Certain observations on the international application	
4.	The International Bureau will communicate this report to designated Offices in accordance with Rules 44bis.3(c) and 93bis.1 but not, except where the applicant makes an express request under Article 23(2), before the expiration of 30 months from the priority date (Rule 44bis .2).	

The International Bureau of WIPO 34, chemin des Colombettes 1211 Geneva 20, Switzerland Facsimile No. +41 22 740 14 35	Date of issuance of this report 18 April 2006 (18.04.2006)
	Authorized officer Nora Lindner Telephone No. +41 22 338 89 65

Form PCT/IB/373 (January 2004)

PATENT COOPERATION TREATY

From the
INTERNATIONAL SEARCHING AUTHORITY



To:

see form PCT/ISA/220

WRITTEN OPINION OF THE INTERNATIONAL SEARCHING AUTHORITY (PCT Rule 43bis.1)

Date of mailing
(day/month/year) see form PCT/ISA/210 (second sheet)

Applicant's or agent's file reference
see form PCT/ISA/220

FOR FURTHER ACTION
See paragraph 2 below

International application No.
PCT/GB2004/004377

International filing date (day/month/year)
13.10.2004

Priority date (day/month/year)
13.10.2003

International Patent Classification (IPC) or both national classification and IPC
G06F15/80

Applicant
CLEARSPEED TECHNOLOGY LIMITED

1. This opinion contains indications relating to the following items:

- ☒ Box No. I Basis of the opinion
- ☐ Box No. II Priority
- ☐ Box No. III Non-establishment of opinion with regard to novelty, inventive step and industrial applicability
- ☒ Box No. IV Lack of unity of invention
- ☒ Box No. V Reasoned statement under Rule 43bis.1(a)(i) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement
- ☐ Box No. VI Certain documents cited
- ☐ Box No. VII Certain defects in the international application
- ☐ Box No. VIII Certain observations on the international application

2. FURTHER ACTION

If a demand for international preliminary examination is made, this opinion will usually be considered to be a written opinion of the International Preliminary Examining Authority ("IPEA"). However, this does not apply where the applicant chooses an Authority other than this one to be the IPEA and the chosen IPEA has notified the International Bureau under Rule 66.1bis(b) that written opinions of this International Searching Authority will not be so considered.

If this opinion is, as provided above, considered to be a written opinion of the IPEA, the applicant is invited to submit to the IPEA a written reply together, where appropriate, with amendments, before the expiration of three months from the date of mailing of Form PCT/ISA/220 or before the expiration of 22 months from the priority date, whichever expires later.

For further options, see Form PCT/ISA/220.

3. For further details, see notes to Form PCT/ISA/220.

Name and mailing address of the ISA:



European Patent Office - P.B. 5818 Patentlaan 2
NL-2280 HV Rijswijk - Pays Bas
Tel. +31 70 340 - 2040 Tx: 31 651 epo nl
Fax: +31 70 340 - 3016

Authorized Officer

Bosch Vivancos, P

Telephone No. +31 70 340-2249



**WRITTEN OPINION OF THE
INTERNATIONAL SEARCHING AUTHORITY**

International application No.
PCT/GB2004/004377

Box No. I Basis of the opinion

1. With regard to the **language**, this opinion has been established on the basis of the international application in the language in which it was filed, unless otherwise indicated under this item.
 - ☐ This opinion has been established on the basis of a translation from the original language into the following language , which is the language of a translation furnished for the purposes of international search (under Rules 12.3 and 23.1(b)).
2. With regard to any **nucleotide and/or amino acid sequence** disclosed in the international application and necessary to the claimed invention, this opinion has been established on the basis of:
 - a. type of material:
 - ☐ a sequence listing
 - ☐ table(s) related to the sequence listing
 - b. format of material:
 - ☐ in written format
 - ☐ in computer readable form
 - c. time of filing/furnishing:
 - ☐ contained in the international application as filed.
 - ☐ filed together with the international application in computer readable form.
 - ☐ furnished subsequently to this Authority for the purposes of search.
3. ☐ In addition, in the case that more than one version or copy of a sequence listing and/or table relating thereto has been filed or furnished, the required statements that the information in the subsequent or additional copies is identical to that in the application as filed or does not go beyond the application as filed, as appropriate, were furnished.
4. Additional comments:

**WRITTEN OPINION OF THE
INTERNATIONAL SEARCHING AUTHORITY**

International application No.
PCT/GB2004/004377

Box No. IV Lack of unity of invention

1. ☐ In response to the invitation (Form PCT/ISA/206) to pay additional fees, the applicant has:
- ☐ paid additional fees.
 - ☐ paid additional fees under protest.
 - ☐ not paid additional fees.
2. ☒ This Authority found that the requirement of unity of invention is not complied with and chose not to invite the applicant to pay additional fees.
3. This Authority considers that the requirement of unity of invention in accordance with Rule 13.1, 13.2 and 13.3 is
- ☐ complied with
 - ☒ not complied with for the following reasons:
see separate sheet
4. Consequently, this report has been established in respect of the following parts of the international application:
- ☒ all parts.
 - ☐ the parts relating to claims Nos.

Box No. V Reasoned statement under Rule 43b/s.1(a)(i) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

1. Statement

Novelty (N)	Yes: Claims	
	No: Claims	1-4,6,8-14,20,22-24,30
Inventive step (IS)	Yes: Claims	
	No: Claims	5,7,15-19,21,25-29,31,32
Industrial applicability (IA)	Yes: Claims	1-32
	No: Claims	

2. Citations and explanations

see separate sheet

Re Item IV

Lack of unity of invention

This Authority considers that there are 3 inventions covered by the claims indicated as follows:

Group I: Claims 1-4,6-10,12-14,20-32

A SIMD processor with local memories in the processing elements.

Group II: Claim 5

A processor with an asynchronous SIMD unit

Group III: Claim 11

Support for nested operations in a SIMD processor

Group IV: Claims 15-19

A multithreaded SIMD processor

The reasons for which the inventions are not so linked as to form a single general inventive concept, as required by Rule 13.1 PCT, are as follows:

The prior art has been identified as document US5717947 (D1) and discloses (column 11, Summary) : a data processor comprising a single control unit (sequencer) and an integral execution unit (scalar engine and vector engine) adapted to operate on both parallel and scalar data under the same instruction set (column 11, line 22 to 24). Document D1 further discloses (figure 2-67) an SIMD part (vector engine) comprising an array of processing elements (PE[0-63]) and a scalar part (scalar engine); that substantially all said instructions are operable on said parallel and/or said scalar data (figures 6-65-1 and 6-65-2) and that each PE has multiple enable bits to support nested conditional code (column 114, lines 35 to 42).

It follows that the following technical features of claims 7, 5, 11 and 15 make a contribution over the prior art and can be considered as special technical features within the meaning of Rule 13.2 PCT:

Group I : the scalar part and the SIMD part differ only in that each element of the SIMD part contains local memory.

**WRITTEN OPINION OF THE
INTERNATIONAL SEARCHING
AUTHORITY (SEPARATE SHEET)**

International application No.

PCT/GB2004/004377

Group II : the scalar part and the SIMD are adapted to operate asynchronously.
Group III : no special technical features have been found.
Group IV : the control unit is adapted to fetch and execute multiple instruction threads simultaneously

The problem solved by these special technical features can therefore be construed as:

Group I : Increase the memory bandwidth of the processor elements in a SIMD processor.
Group II : Increase the instruction level parallelism inside a processor comprising a SIMD unit.
Group IV : Increase the resource utilization of a processor comprising a SIMD unit.

The features of the claims on file which make a contribution over the prior art, i.e. the Special Technical Features, are consequently : local memory on the SIMD processing elements (claim 7), asynchronous execution of the scalar and SIMD parts of the processor (claim 5) and a control unit adapted to fetch and execute multiple instruction threads simultaneously (claim 15)

No special technical feature is present in claims 1-4,6 and 11.

This appears to show lack of corresponding technical effect as well. Consequently, neither the objective problem underlying the subjects of the claimed inventions, nor their solutions defined by the special technical features allow for a relationship to be established between the said inventions, which involves a single general inventive concept.

In conclusion, the groups of claims are not linked by common or corresponding special technical features and define 4 different inventions not linked by a single general inventive concept.

The application, hence does not meet the requirements of unity of invention as defined in Rules 13.1 and 13.2 PCT.

Re Item V

**Reasoned statement with regard to novelty, inventive step or industrial
applicability; citations and explanations supporting such statement**

Reference is made to the following documents:

D1: US5717947

D2: GB2348977

D3: "DLP + TLP Processors for the next generation of Media Workloads"

D4: US5758176

1 Group I (Claims 1-4,6-10,12-14,20-32)

1.1 The present application does not meet the criteria of Article 33(1) PCT, because the subject-matter of claims 1-4, 6, 8-10, 12-14, 20, 22-24 and 30 is not new in the sense of Article 33(2) PCT.

1.1.1 Document D1 discloses a data processor (column 11, line 22 to 24) comprising a single control unit (sequencer) and an integral execution unit (scalar engine and vector engine) adapted to operate on both parallel and scalar data under the same instruction set (ibidem).
Therefore the subject-matter of **claim 1** is not new.

1.1.2 Dependent claims 2-4, 6, 8-10, 12-14, 20, 22-24 and 30 do not contain any features which, in combination with the features of any claim to which they refer meet the requirements of the PCT in respect of novelty, see documents D1 and D2 and the corresponding passages cited in the search report.

1.2 The present application does not meet the criteria of Article 33(1) PCT, because the subject-matter of claims 7, 21, 25-29, 31 and 32 does not involve an inventive step in the sense of Article 33(3) PCT.

1.2.1 Dependent claims 7, 21, 25-29, 31 and 32 do not contain any features which, in combination with the features of any claim to which they refer meet the requirements of the PCT in respect of inventive step, see

documents D1, D2 and the corresponding passages cited in the search report.

2 Group II (Claim 5)

2.1 The present application does not meet the criteria of Article 33(1) PCT, because the subject-matter of claim 5 does not involve an inventive step in the sense of Article 33(3) PCT.

2.1.1 Document D4 discloses a data processor comprising a control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set wherein a scalar and a SIMD part are adapted to operate asynchronously (column 6, lines 50 to 54). The processor of document D4 differs from the processor of claim 5 in that it comprises a single control unit. Using one single control unit is merely one of the available choices in order to implement a processor with a complex execution unit (see for example document D1, column 11, line 22 to 24). Therefore the subject-matter of **claim 5** is not inventive.

3 Group III (Claim 11)

3.1 The present application does not meet the criteria of Article 33(1) PCT, because the subject-matter of claim 11 is not new in the sense of Article 33(2) PCT.

3.2 Document D1 further discloses multiple enable bits to support nested conditional code (column 114, lines 35 to 42), therefore the subject-matter of **claim 11** is not new.

4 Group IV (Claims 15-19)

4.1 The present application does not meet the criteria of Article 33(1) PCT, because the subject-matter of claim 15 is not new in the sense of Article 33(2) PCT.

4.1.1 Document D3 discloses a data processor comprising a control unit and an

integral execution unit adapted to operate on both parallel and scalar data under the same instruction set wherein the control unit is adapted to fetch and execute multiple instruction threads simultaneously (page 221, right column, lines 16 to 20).

Therefore the subject-matter of **claim 15** is not new.

4.2 The present application does not meet the criteria of Article 33(1) PCT, because the subject-matter of claims 16-19 does not involve an inventive step in the sense of Article 33(3) PCT.

4.2.1 Dependent claims 16-19 do not contain any features which, in combination with the features of any claim to which they refer meet the requirements of the PCT in respect of inventive step, see documents D3 and D4 and the corresponding passages cited in the search report.



P.B.5818 - Patentlaan 2
2280 HV Rijswijk (ZH)
☎ (070) 3 40 20 40
FAX (070) 3 40 30 16

Europäisches
Patentamt

European
Patent Office

Office européen
des brevets

Generaldirektion 1

Directorate General 1

Direction générale 1

VIGARS, Christopher, Ian
Haseltine Lake
Redcliff Quay
120 Redcliff Street
Bristol BS1 6HU
GRANDE BRETAGNE



EPO Customer Services

Tel.: +31 (0)70 340 45 00

Date

03.03.06

Reference	Application No./Patent No. 04768907.0 - 2211 PCT/GB2004004377
Applicant/Proprietor Clearspeed Technology PLC	

Entry into the European phase before the European Patent Office

These notes describe the procedural steps required for entry into the European phase before the European Patent Office (EPO). You are advised to read them carefully: failure to take the necessary action in time can lead to your application being deemed withdrawn.

1. The above-mentioned international patent application has been given European application No. **04768907.0**.
2. Applicants **without** a residence or their principal place of business in an EPC contracting state may themselves initiate European processing of their international applications, provided they do so before expiry of the 31st month from the priority date (see also point 6 below).

During the European phase before the EPO as designated or elected Office, however, such applicants must be represented by a professional representative (Arts. 133(2) and 134(1), (7) EPC).

Procedural acts performed after expiry of the 31st month by a professional representative who acted during the international phase but is not authorised to act before the EPO have no legal effect and therefore lead to loss of rights.

Please note that a professional representative authorised to act before the EPO and who acted for the applicant during the international phase does not automatically become the representative for the European phase. Applicants are therefore strongly advised to appoint in good time any representative they wish to initiate the European phase for them; otherwise, the EPO has to send all communications direct to the applicant.

3. Applicants **with** a residence or their principal place of business in an EPC contracting state are not obliged to appoint, for the European phase before the EPO as designated or elected Office, a professional representative authorised to act before the EPO.
However, in view of the complexity of the procedure it is recommended that they do so.
4. Applicants and professional representatives are also strongly advised to initiate the European phase using EPO Form 1200 (available free of charge from the EPO). This however is not compulsory.



5. **To enter the European phase before the EPO**, the following acts must be performed.
(N.B.: Failure validly to do so will entail loss of rights or other adverse legal consequences.)
- 5.1 If the EPO is acting as **designated** or **elected** Office (Arts. 22(1)(3) and 39(1) PCT respectively), applicants must, within 31 months from the date of filing or (where applicable) the earliest priority date:
- a) Supply a translation of the international application into an EPO official language, if the International Bureau did not publish the application in such a language (Art. 22(1) PCT and Rule 107(1)(a) EPC).
If the translation is not filed in time, the international application is deemed withdrawn before the EPO (Rule 108(1) EPC).
This loss of rights is deemed not to have occurred if the translation is then filed within a two-month grace period as from notification of an EPO communication, provided a surcharge is paid at the same time (Rule 108(3) EPC).
 - b) Pay the national basic fee (EUR 160,00) and, where a supplementary European search report has to be drawn up, the search fee (EUR 960,00 ; Rule 107(1)(c) and (e) EPC).
 - c) If the time limit under Article 79(2) EPC expires before the 31-month time limit, pay the designation fee (EUR 75,00) for each contracting state designated (Rule 107(1)(d) EPC).
 - d) If the time limit under Article 94(2) EPC expires before the 31-month time limit, file the written request for examination **and** pay the examination fee (EUR 1430,00 ; Rule 107(1)(f) EPC).
 - e) Pay the third-year renewal fee (EUR 380,00) if it falls due before expiry of the 31-month time limit (Rule 107(1)(g) EPC).
- If the fees under (b) to (d) above are not paid in time, or the written request for examination is not filed in time, the international application is deemed withdrawn before the EPO, or the contracting-state designation(s) in question is (are) deemed withdrawn (Rule 108(1) and (2) EPC). However, the fees may still be validly paid within a two-month grace period as from notification of an EPO communication, provided the necessary surcharges are paid at the same time (Rule 108(3) EPC). For the renewal fee under (e) above, the grace period is **six** months from the fee's due date (Article 86(2) EPC).
- 5.2 If the application documents on which the European grant procedure is to be based comprise more than ten claims, a claims fee is payable within the 31-month time limit under Rule 107(1) EPC for the eleventh and each subsequent claim (Rule 110(1) EPC). The fee can however still be paid within a one-month grace period as from notification of an EPO communication pointing out the failure to pay (Rule 110(2) EPC).
6. If the applicant had a representative during the application's international phase, the present notes will be sent to the representative, asking him to inform the applicant accordingly.
- All subsequent communications will be sent to the applicant, or - if the EPO is informed of his appointment in time - to the applicant's European representative.**



Date

Sheet 3

Application No. 04768907.0

7. For more details about time limits and procedural acts before the EPO as designated and elected Office, see the EPO brochure

How to get a European patent
Guide for applicants - Part 2
PCT procedure before the EPO - "Euro-PCT"

This brochure, the list of professional representatives before the EPO, Form 1200 and details of the latest fees are now all available on the Internet under

<http://www.european-patent-office.org>

RECEIVING SECTION



INTERNATIONAL SEARCH REPORT

Interr ☐ Application No
PCT/GB2004/004377

A. CLASSIFICATION OF SUBJECT MATTER IPC 7 G06F15/80		
According to International Patent Classification (IPC) or to both national classification and IPC		
B. FIELDS SEARCHED		
Minimum documentation searched (classification system followed by classification symbols) IPC 7 G06F		
Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched		
Electronic data base consulted during the international search (name of data base and, where practical, search terms used) EPO-Internal		
C. DOCUMENTS CONSIDERED TO BE RELEVANT		
Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X A	US 5 717 947 A (GALLUP ET AL) 10 February 1998 (1998-02-10) column 11, line 22 - line 24 column 114, line 35 - line 42 -----	1-4, 6-14, 20-32 5
X A	GB 2 348 977 A (* PIXELFUSION LIMITED) 18 October 2000 (2000-10-18) figure 1 figure 3 figure 4 page 16, line 17 - line 25 page 12, line 30 - line 34 page 17, line 15 - page 19, line 30 page 39, line 14 - line 18 ----- -/--	1,15-19 7,21, 25-32
☒ Further documents are listed in the continuation of box C. ☒ Patent family members are listed in annex.		
* Special categories of cited documents : *A* document defining the general state of the art which is not considered to be of particular relevance *E* earlier document but published on or after the international filing date *L* document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified) *O* document referring to an oral disclosure, use, exhibition or other means *P* document published prior to the international filing date but later than the priority date claimed *T* later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention *X* document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone *Y* document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art. *& document member of the same patent family		
Date of the actual completion of the international search 24 June 2005		Date of mailing of the international search report 30/06/2005
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016		Authorized officer Bosch Vivancos, P

Form PCT/ISA/210 (second sheet) (January 2004)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/GB2004/004377

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT		
Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	CORBAL J ET AL: "DLP+TLP processors for the next generation of media workloads" HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 2001. HPCA. THE SEVENTH INTERNATIONAL SYMPOSIUM ON MONTERREY, MEXICO 19-24 JAN. 2001, LOS ALAMITOS, CA, USA, IEEE COMPUT. SOC, US, 19 January 2001 (2001-01-19), pages 219-228, XP010531688 ISBN: 0-7695-1019-1 column 6, line 50 - line 54 -----	1,15-19
X	US 5 758 176 A (AGARWAL ET AL) 26 May 1998 (1998-05-26) abstract figures 2,3,6	1,5
A	-----	16-19

Form PCT/ISA/210 (continuation of second sheet) (January 2004)

INTERNATIONAL SEARCH REPORT

International application No.
PCT/GB2004/004377

Box II Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)

This International Search Report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:
2. ☐ Claims Nos.:
because they relate to parts of the International Application that do not comply with the prescribed requirements to such an extent that no meaningful International Search can be carried out, specifically:
3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

see additional sheet

1. ☐ As all required additional search fees were timely paid by the applicant, this International Search Report covers all searchable claims.
2. ☒ As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.
3. ☐ As only some of the required additional search fees were timely paid by the applicant, this International Search Report covers only those claims for which fees were paid, specifically claims Nos.:
4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this International Search Report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

- ☐ The additional search fees were accompanied by the applicant's protest.
- ☐ No protest accompanied the payment of additional search fees.

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. claims: 1-32

Searched claims

1.1. claims: 1-4,6-10,12-14,20-32

A SIMD processor with local memories in the processing elements

1.2. claim: 5

A processor with an asynchronous SIMD unit

1.3. claim: 11

Support for nested operations in a SIMD processor

1.4. claims: 15-19

A multithreaded SIMD processor

INTERNATIONAL SEARCH REPORT

International Application No
PCT/GB2004/004377

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5717947 A	10-02-1998	CN 1107983 A ,C	06-09-1995
		EP 0619557 A2	12-10-1994
		JP 7073149 A	17-03-1995
		US 6085275 A	04-07-2000
		US 5742786 A	21-04-1998
		US 5752074 A	12-05-1998
		US 5600846 A	04-02-1997
		US 5664134 A	02-09-1997
		US 5706488 A	06-01-1998
		US 5598571 A	28-01-1997
		US 5754805 A	19-05-1998
		US 5572689 A	05-11-1996
		US 5737586 A	07-04-1998
		US 5734879 A	31-03-1998
		US 5548768 A	20-08-1996
		US 5537562 A	16-07-1996
		US 5559973 A	24-09-1996
		US 5805874 A	08-09-1998
		US 5600811 A	04-02-1997
		US 5790854 A	04-08-1998
GB 2348977 A	18-10-2000	AU 3829500 A	14-11-2000
		EP 1181648 A1	27-02-2002
		WO 0062182 A2	19-10-2000
		JP 2002541586 T	03-12-2002
		US 2002174318 A1	21-11-2002
US 5758176 A	26-05-1998	NONE	

Form PCT/ISA/210 (patent family annex) (January 2004)

(19) World Intellectual Property
Organization
International Bureau



(43) International Publication Date
28 April 2005 (28.04.2005)

PCT

(10) International Publication Number
WO 2005/037326 A3

(51) International Patent Classification⁷: **G06F 15/80**

(21) International Application Number:
PCT/GB2004/004377

(22) International Filing Date: 13 October 2004 (13.10.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0323950.6 13 October 2003 (13.10.2003) GB
0409815.8 30 April 2004 (30.04.2004) GB

(71) Applicant (for all designated States except US): **CLEAR-SPEED TECHNOLOGY PLC** [GB/GB]; 3110 Great Western Court, Hunts Ground Road, Stoke Gifford, Bristol BS34 8HP (GB).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **STUTTARD, David**

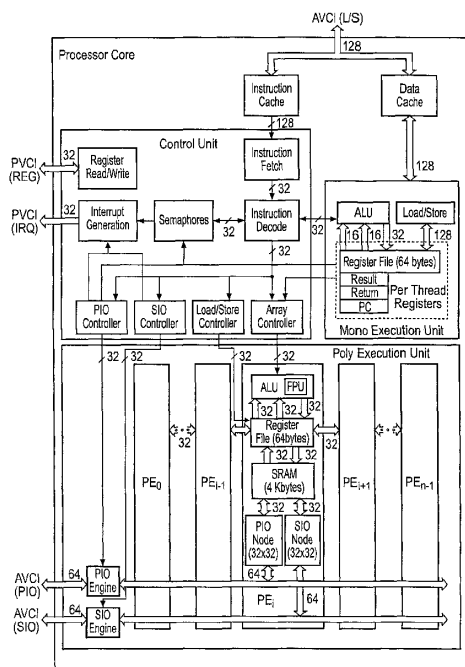
[GB/GB]; 28 St Georges Avenue, St George, Bristol BS5 8DD (GB). **WILLIAMS, David** [GB/GB]; 20 Cousins Way, Emersons Green, Bristol BS16 7DG (GB). **PACKER, James** [GB/GB]; 43 Whitesfield Road, Nailsea, Bristol BS48 2DY (GB). **DAVIDSON, Colin** [GB/GB]; 46 Magdelene Street, Glastonbury, Somerset BA6 9EJ (GB). **HICKEY, Neil** [GB/GB]; 34 Casson Drive, Stoke Park, Bristol BS16 1WR (GB). **DAY, Timothy** [GB/GB]; 33 Rosebery Avenue, St. Werburghs, Bristol BS2 9TW (GB).

(74) Agent: **VIGARS, Christopher, Ian**; Haseltine Lake, Redcliff Quay, 120 Redcliff Street, Bristol BS1 6HU (GB).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG,

[Continued on next page]

(54) Title: **UNIFIED SIMD PROCESSOR**



(57) Abstract: A data processor comprises a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set. The execution unit comprises a SMO part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data. Substantially all the instructions are operable on the parallel and/or the scalar data. Preferably, both the scalar part and the SIMD part have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the execution unit; SHVM variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory. The scalar part, the SIMD part, and I/O means are adapted to be operated in parallel by the same instruction stream.

WO 2005/037326 A3



PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

- *with international search report*
- *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

(88) Date of publication of the international search report:

25 August 2005

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

Document made available under the Patent Cooperation Treaty (PCT)

International application number: PCT/GB04/004377

International filing date: 13 October 2004 (13.10.2004)

Document type: Certified copy of priority document

Document details: Country/Office: GB
Number: 0323950.6
Filing date: 13 October 2003 (13.10.2003)

Date of receipt at the International Bureau: 02 March 2005 (02.03.2005)

Remark: Priority document submitted or transmitted to the International Bureau in compliance with Rule 17.1(a) or (b)



World Intellectual Property Organization (WIPO) - Geneva, Switzerland
Organisation Mondiale de la Propriété Intellectuelle (OMPI) - Genève, Suisse



PCT/GB 2004/004377



INVESTOR IN PEOPLE

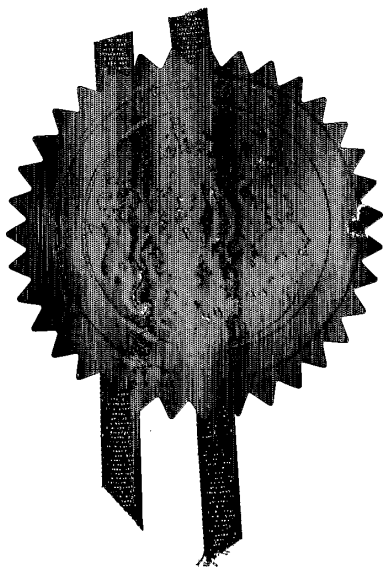
The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.



Signed

Dated 10 February 2005



Patents Act 1977
(Rule 16)

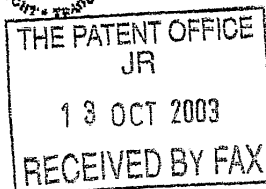


1/77

Request for grant of a patent

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form)

13 OCT 2003



The Patent Office

Cardiff Road
Newport
Gwent NP9 1RH

1.	Your reference	P100390GB00/CIV		
2.	Patent application number (The Patent Office will fill in this)	0323950.6		
3.	Full name, address and postcode of the or of each applicant (underline all surnames)	ClearSpeed Technology Limited 3110 Great Western Court Hunts Ground Road Stoke Gifford Bristol BS34 8HP		
	Patents ADP number (if you know it)			
	If the applicant is a corporate body, give the country/state of its incorporation	8484933001		
4.	Title of the invention	Unified SIMD Processor		
5.	Full name of your agent (if you have one)	Haseltine Lake		
	"Address for service" in the United Kingdom to which all correspondence should be sent (including the postcode)	Imperial House 15-19 Kingsway London WC2B 6UD		
	Patents ADP number (if you know it)	34001		
6.	If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and (if you know it) the or each application number	Country	Priority application number (if you know it)	Date of filing (day/month/year)
7.	If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application	Number of earlier application	Date of filing (day/month/year)	
8.	Is a statement of inventorship and of right to a grant of patent required in support of this request? (Answer "Yes" if: a) any applicant named in part 3 is not an inventor, or b) there is an inventor who is not named as an applicant, or c) any named applicant is a corporate body. See note (d))	Yes		

Patents Form 1/77

0082668 13-Oct-03 04:25

Patent Form 1/77

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document.

Continuation sheets of this form

Description	21
Claims(s)	
Abstract	
Drawing(s)	6

10. If you are also filing any of the following, state how many against each item.

Priority documents

Translations of priority documents

Statement of inventorship and right to a grant of patent (Patents Form 7/77)

Request for preliminary examination and search (Patents Form 9/77)

Request for substantive examination (Patents Form 10/77)

Any other documents (please specify)

11. I/We request the grant of a patent on the basis of this application.

Haseltine Lake, Agents for the Applicants

Signature

Date

Haseltine Lake

13 October 2003

12. Name and daytime telephone number of person to contact in the United Kingdom
- Mr Chris Vigers [0117] 910 3200

Warning

After an application for a patent has been filed, the Comptroller of the Patent Office will consider whether publication or communication of the invention should be prohibited or restricted under Section 22 of the Patents Act 1977. You will be informed if it is necessary to prohibit or restrict your invention in this way. Furthermore, if you live in the United Kingdom, Section 23 of the Patents Act 1977 stops you from applying for a patent abroad without first getting written permission from the Patent Office unless an application has been filed at least 6 weeks beforehand in the United Kingdom for a patent for the same invention and either no direction prohibiting publication or communication has been given, or any such direction has been revoked.

Notes

- If you need help to fill in this form or you have any questions, please contact the Patent Office on 0645 500505.
- Write your answers in capital letters using black ink or you may type them.
- If there is not enough space for all the relevant details on any part of this form, please continue on a separate sheet of paper and write "see continuation sheet" in the relevant part(s). Any continuation sheet should be attached to this form.
- If you have answered 'Yes' Patents Form 7/77 will need to be filed.
- Once you have filled in the form you must remember to sign and date it.
- For details of the fee and ways to pay please contact the Patent Office.

Patents Form 1/77

0082668 13-Oct-03 04:25

Unified SIMD Processor

Background to the Invention

- The invention consists of a significant improvement to a well known type of parallel processor. Single-Instruction Multiple-Data (SIMD) architectures are widely used in a number of computing applications. They have the advantage of high performance, low power consumption and easy scalability. A SIMD processor differs from the more commonly used multi-processor system (where a number of processors, each independently executing their own program code, are used together to solve a problem) in that it executes a single stream of instructions but each instruction operates on multiple data items in parallel.
- There are a range of SIMD processor architectures. These range from an *intelligent memory* type of approach where each *column* of a memory array has a simple 1-bit Arithmetic Logic Unit (ALU) through to more complex architectures where a, usually smaller, array of more complex processing elements are fed with a single instruction stream. They have, however, been difficult to program in the past.
- In all these cases the array of SIMD processing elements are fed with instructions and data by a dedicated controller. Often the controller is the largest and most complex part of the design because of the need to match the data processing rate achieved by the SIMD array. The array controller is not itself 'intelligent' and is, in turn, controlled by a general purpose processor which also runs the bulk of the application software. This *host* processor will execute most of the application code; it will send data to be processed to the SIMD array with an indication of the function to be performed, typically a pointer to the start of a sequence of SIMD array instructions or microinstructions.
- This introduces two programming problems. Firstly, the array controller can only be programmed at a very low level: a series of microcode, or possibly assembler, instructions to be executed by the array. This is because the array controller is not designed to work with a compiler. Secondly, this programming is completely separated from the main application code running on the host processor. This may be 'wrapped up' by providing C functions or C++ classes which makes them appear more integrated and reduces the complexity of passing control between the host processor and the SIMD array. Also, in the simplest form, the host processor and the SIMD array will have completely separate memory spaces for code and data. Any of the standard shared memory techniques can be used to address this with all the resulting problems: complex arbitration, cache

coherency issues, contention for memory bandwidth, inter-processor synchronization, greater cost, etc.

A preferable solution is to provide a fully integrated architecture where the same processor can transparently process both *scalar* and SIMD operations in a fully integrated fashion. This means
5 that a single program can be written, in a standard high-level language such as C, which includes all parts of the application – both ‘normal’ (scalar) data and parallel data processing.

Prior Art

The basic SIMD processing model has been in use for many years. Some of the earliest examples are the ILLIAC IV developed in 1972, which had 64 processors with floating point and memory,
10 and Goodyear’s STARAN (1975) which was a 1-bit architecture. Many variants have been developed since then.

There is also a recent trend to put small-scale SIMD extensions in standard processor architectures. This is exemplified by the Intel MMX and SSE extensions to the x86 architecture. This allows, for example, the values in two 32-bit registers to be added as if each register
15 contained four independent 8-bit values. This is not considered relevant to the current discussion for a number of reasons:

- the SIMD processing is done on a very small scale;
- only a limited range of SIMD functionality is provided;
- there are distinct instructions for operating on SIMD data;
- 20 • the same hardware is used for SIMD and for scalar operations (although separate registers may be provided for the SIMD operands)

The idea of providing indexed addressing modes in a SIMD processor is disclosed in our copending patent application GB9908225.7. This invention extends that idea by using a more flexible implementation of that idea in combination with a more powerful array controller to
25 provide a unified processor architecture.

Summary of the Invention

Applicant’s approach, after many years of experience with developing systems based on SIMD coprocessors, was to create a new architecture where instructions are decoded and despatched to the appropriate part of the execution unit: either the scalar (mono) execution unit or the SIMD
30 (poly) execution unit. Instruction execution on these are decoupled so that, for example, a mono instruction followed by a poly instruction will execute concurrently. The support for mono and poly data is, as far as possible, identical – this makes it practical to write a compiler to target the processor. A simple extension to the programming language allows data to be marked as either being processed on the mono or poly execution units. This greatly simplifies the task of

-3-

programming the processor, especially as it is not normally necessary for the program to change depending on the size of the SIMD array: poly data is automatically distributed across the array.

This means a single program runs on the processor which does both mono and poly operations as required – typically it will do both at the same time. A single C program can be written which

5 manages program control flow, operates on mono data, and operates on poly data.

To support this, and further simplify programming, this unity of the architecture is reflected in the instruction set and programming model. The processing elements in the SIMD array and the mono execution unit have a common instruction set. For instance, an add instruction can be used with operands which are either poly registers or mono registers (or immediate values); the

10 appropriate execution unit will execute the instruction. Mono and poly operands can also be mixed to allow, for example, a single mono value to be added to a poly register which holds a different value on every processing element.

It is appropriate to note that this does not require the mono execution unit to have exactly the same micro-architecture as the poly execution unit. For example, the poly processing elements

15 could have an 8-bit ALU, while the mono execution unit has a 32-bit ALU.

The most significant features contributing to the novelty and inventiveness of this invention include the following concepts:

- The processor provides a unified programming model: a single program runs on a processor providing both scalar and SIMD operations.
- 20 • This is supported with a unified instruction set: e.g. an add instruction can work with either mono or poly operands (or a mixture of the two)
- Poly operations can freely use mono registers in expressions, and more importantly for addressing memory.
- Having a full range of addressing modes and completely separate memory blocks in each
- 25 PE make it simple to support poly data structures and poly pointers on the SIMD array (previous SIMD architectures either provide a single address to all PEs or use sequences of instructions to construct an address in memory).
- Mono and poly operations are of equal standing: wherever possible, all instructions can be used with either mono or poly operands (there are some exceptions because a small set of
- 30 operations, mainly for control, only make sense for one or the other).
- Similarly, the compiler can mix poly and mono data in expressions and control structures. In particular mono and poly pointers to mono or poly data, and all combinations of these, can be supported.

- The mono and poly execution units are loosely coupled so that their operation can be overlapped. Compilers are extremely good at scheduling instructions to take advantage of this flexibility - maximizing the efficiency of the system.
- There are some compiler optimizations specific to this architecture: for example it may be better, in some instances, to move an operation that would normally be done on the mono execution unit on to the poly execution unit - either because it can then be overlapped with other operations on the mono execution unit or because the result is needed on the poly execution unit anyway. Again, the compiler can detect such cases and make the appropriate optimizations.
- A common code and data space is available to both mono and poly operations making it easy to generate code for, and share data between, the two execution units.
- The overall architecture is suitable for targeting by a high level language compiler - again for both mono and poly operations.

Brief Description of the Drawings

- The invention will be described with reference to the following drawings, in which:
- Figure 1 shows a prior art SIMD processor;
 - Figure 2 shows the architecture in accordance with the present invention;
 - Figure 3 is a comparison between RISC and MTAP architecture;
 - Figure 4 is a schematic diagram of an MTAP execution unit;
 - Figure 5 indicates schematically the execution of an instruction;
 - Figure 6 is a graphical representation of the results of running an example program using the present invention;
 - Figure 7 is a graph indicating the power efficiency of the MTAP architecture;
 - Figure 8 is a block schematic diagram depicting an exemplary MTAP processor; and
 - Figure 9 is a block diagram of a specific implementation of the MTAP architecture as applied to an evaluation chip.

Detailed Description of the Illustrated Embodiments

The design of this processor architecture was driven by the desire to make it practical and efficient to program in a high level language. In order to make this possible a number of features are required. First, the overall programming model needs to be simple and 'regular' - i.e. the same operations and addressing modes can be applied to all data types.

The execution unit for mono data is based on well known principles used in many RISC processors. This makes it simple for a compiler to target and easy for a user to understand.

-5-

The functionality supported by each PE is then made as similar as possible to the mono execution unit: each PE has an ALU, a register file and memory, and supports a range of addressing modes for transferring data between memory and registers.

It is not necessary for the PEs to be identical to the mono execution unit in every detail: e.g. the number of registers and the width and complexity of the ALU are likely to be different in practice. The poly execution unit gets its performance largely from the number of PEs brought to bear on the problem rather than the processing power of the individual PEs. To balance this, the mono execution unit is likely to require a wider and more complex ALU. This is reasonable because this is only instantiated once while the PEs are replicated many times and need to be efficiently implemented.

The number of clock cycles required to execute an instruction on the mono and poly execution units is highly variable. Both will execute some instructions in a single cycle. But some instructions could take several cycles on the mono execution unit or on the poly execution unit. It is therefore essential that the two execution units are only loosely coupled rather than operating in lock-step; this allows them to overlap their execution of instructions. The hardware will synchronize their operations when necessary: e.g. when they need to access a shared resource, or one execution unit requires data from the other.

Transfers between the registers and memory, and other I/O transfers, can take place concurrently with ALU operations. To simplify the task for the compiler, these interactions are all interlocked using register scoreboarding (see our copending patent application GB 9908203.4).

An important aspect of the PE architecture to support compiled code is the existence of a full set of addressing modes: direct, indirect and indexed. Indirect and indexed addressing by the PEs is described in the previously mentioned copending patent application GB 9908225.7.

In a high level language compiler for this architecture, it is only necessary to introduce a simple modification; the keyword 'poly' used in a declaration identifies data that is distributed across the SIMD array. Similar methods have been used before, for example the 'shape' keyword used in C*, a language used for programming the "Connection Machine". However, it is only with the present architecture that poly variables become 'first class' objects - i.e. they can be used everywhere that 'normal' mono data can be: in expressions, as function arguments, in conditional statements, etc. Pointers to both mono and poly data are allowed, as are pointers of both mono and poly types. Furthermore, poly variables can be mixed quite freely with mono data in all these contexts. This provides a very simple and regular programming model for the user.

Note that it would also be possible for a compiler to detect the parallelism in the program or data automatically: This is frequently done with 'vectorizing' Fortran compilers used with

0082668 13-Oct-03 04:25

supercomputers, for example. The same techniques could be applied to this architecture. The architecture will now be described in more detail.

The Multi-Threaded Array Processing (MTAP) architecture has been developed to address a number of problems in high performance, high data rate processing.

- 5 A number of applications are characterized by very high data rates, flexible processing requirements and hard real-time constraints. We use the term *data flow processing* to describe this class of problem.

The MTAP processor delivers on the three primary requirements for data flow applications:

1. It directly addresses the data bandwidth, and has a clear scalability path for future
10 requirements.
2. It provides the raw horsepower for the processing functions required, on the maximum data rate that the system will encounter. That processing power can scale with increasing function demands.
3. It stores the data close to the processing core to maximize bandwidth and minimize latency.

15 **MTAP Overview**

The MTAP architecture defines a family of embedded processors with parallel data processing capability. Figure 1 compares a standard processor and the MTAP architecture. As can be seen, the MTAP processor has a standard, RISC-like, control unit with instruction fetch, caches and I/O mechanisms. This is coupled to a highly parallel execution unit which provides the
20 performance and scalability of the MTAP architecture.

The processor is designed in a highly modular fashion which allows many details of a specific implementation to be easily customized for the target application.

To simplify the integration of the processor into a variety of systems, the processor can be also configured to be big or little-endian.

25 **Control unit**

The control unit fetches, decodes and dispatches instructions to the execution units. The processor executes a fairly standard, three operand instruction set.

The control unit also provides hardware support for multi-threaded execution, allowing fast swapping between multiple threads. The threads are prioritized and are intended primarily to
30 support efficient overlap of I/O and compute. This can be used to hide the latency of external data accesses.

The MTAP processor can also include instruction and data caches to minimize the latency of memory accesses. The size and type of these is a configurable option. Alternatively, the caches may be replaced with SRAM to provide a complete embedded solution.

-7-

The control unit also includes a control port which is used for initializing and debugging the processor; it includes support for breakpoints, single stepping and the examination of internal state.

Execution unit

- 5 The execution unit consists of a number of *processing elements* (PEs). This allows it to process data elements in parallel. Each PE consists of an ALU, registers, memory and I/O.

The number of PEs in a processor core is a configurable parameter which allows performance to be scaled to meet the needs of the application. The execution unit can be thought of as two, largely independent, parts (see Figure 4).

- 10 One PE forms the *mono* execution unit; this is dedicated to processing mono (i.e. scalar or non-parallel) data. The mono execution unit also handles program flow control such as branching and thread switching. The rest of the PEs form the *poly* execution unit which processes parallel (poly) data.

- The poly execution unit may consist of tens, hundreds or even thousands of PEs. This array of
15 PEs operates in a synchronous manner, similar to SIMD, where every PE executes the same instruction on its piece of data. Each PE also has its own independent local memory; this provides fast access to the data being processed. For example, one PE at 400 MHz has a memory bandwidth of 1.6 Gbytes/s. An array with 256 such PEs has an aggregate bandwidth of over 400 Gbytes/s, with single cycle latency.

- 20 The number of registers and the amount of memory in each PE are configurable options.

Input / output

- The MTAP processor core has two basic I/O mechanisms. The first, *Programmed I/O* (PIO), is the normal mechanism used for accessing memory external to the MTAP core: it supports random accesses to variable sized data by the PEs. The second mechanism is *Streaming I/O*
25 (SIO) which allows chunks of contiguous data to be streamed directly into the memory within PEs.

Each of these supports a variety of addressing modes; these are described in more detail later.

Programming model

- From a programmer's perspective, the MTAP processor appears as a single processor running a
30 single C program. This is very different from some other parallel processing models where the programmer has to explicitly program multiple independent processors, or can only access the processor via function calls or some other indirect mechanism.

The MTAP processor executes a single instruction stream; each instruction is sent to one of the functional units: this may be the mono or poly execution unit, or one of the I/O controllers. The processor can despatch an instruction on every cycle.

For multi-cycle instructions, the operation of the functional units can be overlapped. So, for example, an I/O operation can be started with one instruction and on the next cycle the mono execution unit could start a multiply instruction (which requires several cycles to execute). While these operations are proceeding, the poly execution unit can continue to execute instructions.

The main change from programming a standard processor is the concept of operating on parallel data. Data to be processed is assigned to variables which have an instance on every PE and are operated on in parallel: we call this *poly* data. This can be thought of as a data vector which is distributed across the array.

Variables which only need to have a single instance (e.g. loop control variables) are known as *mono* variables – they behave exactly like normal variables on a sequential processor.

Applicant provides a compiler which uses a simple extension to standard C to identify data which is to be processed in parallel. The new keyword *poly* is used in a declaration to define data which exists, and is processed, on every PE in the poly execution unit.

Example

To give a feel for the way the MTAP processor is programmed, the following represents a fragment of code which will calculate 64 values of a sine function across the PE array in a single operation.

```
#include <fnext.h>
#include <math.h>
#define PI 3.1415926535897932384
#define NUMBER_OF_PES 64

poly float angle, sine;
poly int pe;

/* get PE number: 0...n-1 */
30 pe = get_penum();
/* convert to an angle in range 0 to Pi */
angle = pe * PI / NUMBER_OF_PES;
/* calculate sine of angle on each PE */
sine = sinp(angle);
```

This code uses the library function `get_penum()` to get a unique value in the variable `pe` on each PE. This is scaled to give a range of values for `angle` between 0 and π across the PEs.

Finally, the library function `sinp()` is called to calculate the sine of these values on all PEs simultaneously. The `sinp` function is the poly equivalent of the standard sine function; it takes a

5 poly argument and returns the appropriate value on every PE.

The results of running this program on a 64-PE MTAP processor are shown in a graphical representation in Figure 6.

Performance

10 The performance of an MTAP core with 64 PEs, on a 0.13μ process and running at 400 MHz is shown in Table 1. The columns show the performance achieved with the basic PE, the PE with an additional Multiply-Accumulate (MAC) unit, and the PE with an FPU extension.

Table 1 Performance Comparison

	Base Architecture	8-bit MAC	FPU
MIPS	25,600	25,600	25,600
MMACS ($\times 10^6$)		25,600	25,600
MFLOPS			51,200
Memory Bandwidth (Gbytes / s)	100	100	100
Inter-PE Bandwidth (GBytes / s)	50	50	50
Relative Area	1.0	1.1	1.3

15 However, the core can be scaled well beyond this number of PEs, to 256 or even thousands of PEs. It also scales down efficiently to lower numbers of PEs, or lower clock rates, enabling low cost and low power systems to be built.

Figure 7 shows how performance and power scale with the number of PEs. By choosing the number of PEs and the clock frequency, the designer has a great deal of flexibility to tune

20 performance, bandwidth, cost and power for the needs of a given application.

Architecture Details

The following sections describe the MTAP processor in more detail. A block diagram of the processor is shown in Figure 8. This is a conceptual view, which reflects the programming model. In practice, the control and mono execution units form a tightly-coupled pipeline which provides overall control of the processor.

5 *Interfaces*

The processor has a number of external interfaces. The widths of each of these are configurable. All these interfaces currently use the Virtual Component Interface Standard defined by the VSI Alliance.

There are three categories of interface:

10 *Mono data & instructions*

This is a single *Advanced VCI* (AVCI) interface which is used for mono loads and stores, and for instruction fetching.

Poly data

This uses one or more AVCI interfaces for PIO and SIO data. The number of physical interfaces corresponds to the number of PIO and SIO channels implemented.

Control

There are two control interfaces which use the *Peripheral VCI* (PVCI) protocol. One is used for initialization and debug. The other allows the MTAP processor to generate interrupts to the host system.

20 *Instruction set*

The processor has a fairly standard RISC-like instruction set. Most instructions can operate on mono or poly operands and are executed by the appropriate part of the execution unit. Some instructions are only relevant to either the mono or poly execution unit, for example all program flow control is handled by the mono unit.

25 The instruction set provides a standard set of functions on both mono and poly execution units:

- Integer and floating point adds, subtracts, multiplies, divides
- Logical operations: and, or, not, xor
- Arithmetic and logical shifts
- Data comparisons: equal, not equal, greater than, less than, etc.
- 30 • Data movement between registers
- Loads and stores between registers and memory

To give a feel for the nature of assembly code for the MTAP processor, a few lines of code are shown below. This simple example loads a value from memory into a mono register, gets the PE

-11-

number into a register on each PE and then adds these two values together producing a different result on every PE.

```
ld 0:m4, 0x3000    // load mono reg 0 from mem
penum 8:p4          // PE number into poly reg
5 add 4:p4, 0:m4, 8:m4 // add; result in reg 4
```

Control unit

The control unit fetches instructions from memory, decodes them and despatches them to the appropriate functional unit.

The controller includes a scheduler to provide hardware support for multi-threaded code. This is a vital part of the architecture for achieving the performance potential of the MTAP processor. Because of the highly parallel architecture of the poly execution unit, there can be significant latencies if all PEs need to read or write external data.

When part of an application stalls because it is waiting for data from external memory, the processor can switch to another code thread that is ready to run. This serves to hide the latency of accesses and keep the processor busy.

The number of threads is a configurable option. The threads are prioritized: the processor will run the highest priority thread that is ready to run; a higher priority thread can pre-empt a lower priority thread when it becomes ready to run. Threads are synchronized – with each other and with hardware, such as I/O engines – via semaphores.

In the simplest case of multi-threaded code, a program would have two threads: one for I/O and one for compute. By *pre-fetching* data in the I/O thread, the programmer (or the compiler) can ensure the data is available when it is required by the execution units and the processor can run without stalling.

Execution units

This section describes the common aspects of the poly and mono execution units.

ALU operations

Instructions for arithmetic and logical operations are provided in several versions:

- Various sizes: 1, 2, 3 and 4 bytes, and combinations of sizes
- Various data types: signed and unsigned integers, and floating-point
- 30 • Mixes of signed/unsigned operations

Status register

Associated with the ALU is a status register; this contains five status bits that provide information about the result of the last ALU operation. When set, these bits indicate;

- Most significant bit set

0082668,13-Oct-03,04:25

- Carry generated
- Overflow generated
- Negative result
- Zero result

5 Registers

To support operations on data of different widths, the registers in the PEs can be accessed very flexibly. The register files are best thought of as an array of bytes which can be addressed as registers of 1 to 8 bytes wide.

- 10 The mono and poly registers are addressed in a consistent way using byte addresses and widths specified in bytes. The mono register file is 16 bits wide and so all addresses and widths must be a multiple of 2. There are no alignment restrictions on poly register accesses.

Addressing modes

- 15 Load and store instructions are used to transfer data between memory and registers. In the case of the mono execution unit, these transfer data to and from memory external to the MTAP processor.

Poly loads and stores transfer data between the PE register file and PE memory. Data is transferred between the PEs and external memory using the I/O functions described later.

There are three addressing modes for loads and stores which can be used for both mono and poly data. These are:

20 Direct

The address to be read/written is specified as an immediate value.

Indirect

The address is specified in a register.

Indexed

- 25 The address is calculated from adding an offset to a base address in a register. The offset must be an immediate value.

Conditional code

The main difference between code for the mono and the poly execution units is the handling of conditional execution.

- 30 The mono unit uses conditional jumps to branch around code, typically based on the result of the previous instructions. This means that mono conditions affect both mono and poly operations. This is just like a standard RISC architecture.

-13-

The poly unit uses a set of enable bits (described in more detail below) to control whether each PE will have its state changed instructions it executes. This provides per-PE predicated operation.

The enable state can be changed based on the result of a previous operation.

The following sections describe the architectural features of the two execution units in more detail.

Mono execution unit

The mono execution unit is a 16-bit processing element consisting of:

- A 16-bit ALU with optional extensions
- A register file of configurable size
- Status and control registers

The mono ALU extensions include a multiplier, a barrel shifter and a normalizer, which is important for accelerating software implementations of floating point operations.

As well as handling mono data, the mono unit is responsible for program flow control (branching), thread switching and other control functions. The mono execution unit also has overall control of I/O operations of poly data. Results from these operations are returned to a register in the mono unit.

Conditional execution

The mono execution unit handles conditional execution in the same way as a traditional processor. A set of conditional and unconditional jump instructions use the result of previous operations to jump over conditional code, back to the start of loops, etc.

Multi-threaded execution

The MTAP processor supports several hardware threads. There is a hardware scheduler in the control unit and the mono execution unit maintains multiple banks of critical registers for fast context switching.

The threads are prioritized (0 being highest priority). Control of execution between the threads is performed using semaphores under programmer control. Higher priority threads will only yield to lower priority threads when they are stalled on yielding instructions (such as semaphore wait operations). Lower priority threads can be pre-empted at any time by higher priority ones.

Semaphores are special registers that can be incremented or decremented with atomic (non-interruptible) operations called signal and wait. A *signal* instruction will increment a semaphore.

A *wait* will decrement a semaphore unless the semaphore is 0, in which case it will stall until the semaphore is signalled by another thread. Semaphores can also be accessed by hardware units (such as the I/O controllers) to synchronize these with software.

Poly execution unit

0082668 13-Oct-03 04:25

The poly execution unit is an array of Processing Elements (PEs). Each PE in the poly execution unit consists of:

- An 8-bit ALU with optional extensions such as a multiply-accumulate (MAC) unit
- A register file of configurable size
- 5 • Status and enable registers
- A block of memory of configurable size
- An inter-PE communication path
- One or more I/O channels

Load and store instructions move data between a PE's register file and memory, while the ALU operates on data in the register file. Data is transferred in to, and out of, the PE's memory using I/O instructions.

ALU

The poly ALU is used for performing arithmetic and logical operations on values held in the PE register file.

- 15 While the ALU is only 8 bits wide, instructions exist for multi-byte arithmetic which is handled by iteration. ALU extensions exist to accelerate functions for floating point, DSP, etc. For example, an optional integer multiply-accumulate unit (MAC) can be included in the ALU. This can deliver an 8 x 8 bit MAC result every cycle, or a 16 x 16 bit MAC every four cycles. The accumulated result can be up to 64 bits wide.

- 20 The standard ALU includes basic hardware support to accelerate floating point operations.

Conditional behavior

- The SIMD nature of the PE array prohibits each PE having its own branch unit (branching being handled by the mono execution unit). Instead, each PE can control whether its state should be updated by the current instruction by *enabling* or *disabling* itself; this is rather like the predicated instructions in some RISC CPUs.
- 25

Enable state

- A PE's enable state is determined by a number of bits in the enable register. If all these bits are set to one, then a PE is *enabled* and executes instructions normally. If one or more of the enable bits is zero, then the PE is *disabled* and most instructions it receives will be ignored (instructions on the enable state itself, for example, are not be disabled).
- 30

The enable register is treated as a stack, and new bits can be pushed onto the top of the stack allowing nested predicated execution. The result of a test, either a 1 or a 0, is pushed onto the enable stack. This bit can later be popped from the top of the stack to remove the effect of that condition. This makes handling nested conditions and loops very efficient. Note that, although

-15-

the enable stack is of fixed size, the compiler handles saving and restoring the state automatically, so there are no limitations on compiled code. When programming at the assembler level, it is the programmer's responsibility to manage the stack.

Instructions

- 5 Conditional execution on the poly execution unit is supported by a set of poly conditional instructions: *if*, *else*, *endif*, etc. These manage the enable bits to allow different PEs to execute each branch of an *if...else* construct in C, for example. These also support nested conditions by pushing and popping the condition value on the enable stack.

As a simple example, consider the following code fragment:

```
10 // disable PEs where reg 32 is non-zero
   if.eq 32:p1, 0 // push result onto stack
   // increment reg 8 on enabled PEs
   add 8:p4, 8:p4, 1
   // return all PEs to original enable state
15 endif // pop enable stack
```

Here, the initial *if* instruction compares the two operands on each PE. If they are equal it pushes 1 onto the top of the enable stack – this leaves those PEs enabled if they were previously enabled and disabled if they were already disabled. If the two operands are not equal, a 0 is pushed onto the stack – this disables the corresponding PEs.

- 20 The following *add* instruction is sent to all PEs, but only acted on by those that are still enabled. Finally, the *endif* instruction *pops* the enable stack, returning all PEs to their original enable state.

Forced loads and stores

- Poly loads and stores are normally predicated by the enable state of the PE. However, because 25 there are instances where it is necessary to load and store data regardless of the current enable state, the instruction set includes *forced* loads and stores. These will change the state of the PE even if it is disabled.

I/O mechanisms

- There are two I/O mechanisms provided for transferring data between PE memory and devices 30 outside the MTAP core. *Programmed I/O* (PIO) extends the load/ store model: it is used for transfers of small amounts of data between PE memory and external memory. *Streaming I/O* (SIO) is used to efficiently transfer contiguous chunks of data in and out of PE memory.

The number of PIO and SIO channels in a processor is a configurable parameter. A processor will always have at least one PIO channel as this is required for normal program execution.

0082668 13-Oct-03 04:25

Multiple I/O channels can run simultaneously.

I/O architecture

The I/O systems consist of three parts: Controller, Engine and Node.

Controller

- 5 The PIO and SIO controllers decode I/O instructions and coordinate with the rest of the control unit and the mono processor. The controllers synchronize with software threads via semaphores.

Engine

The I/O engines are basically DMA engines which manage the actual data transfer.

There is a Controller and Engine for each I/O channel. A single Controller can manage several

- 10 I/O Engines.

Node

There is an I/O Node in each PE. The I/O Engine activates each Node in turn allowing to serialize the data transfers. The Nodes provide buffering of data to minimize the impact of I/O on the performance of the PEs.

- 15 **Programmed I/O (PIO)**

PIO is closely coupled to program execution and is the normal way for the processor to transfer data to and from the outside world (e.g. external memory, hardware accelerators or a host processor).

The PIO mechanism provides a number addressing modes:

20

Direct addressed

Each PE provides an external memory address for its data. This provides random access to data.

Strided

- 25 The external memory address is incremented for each PE. In each case, the size of data transferred to each PE is the same.

- When multiple PEs are accessing memory then the transfers can be *consolidated* so as to perform the minimum number of external accesses. So, for example, if half the processors are reading one location and the other half reading another, then only two memory reads would be performed. In fact, consolidation can be better than that: because the bus transfers are *packetized*, even transfers
30 from nearby addresses can be effectively consolidated.

Streaming I/O (SIO)

SIO is used for streaming high-bandwidth data directly to and from PE memory. This is less flexible but very efficient for transferring blocks of data in and out of the system. This is

typically used to stream data to and from memory mapped I/O devices. Each PE can transfer a different size block of data.

Streaming I/O can be made very efficient in a number of ways. For example, multi-threaded code allows data I/O and compute to be fully overlapped. Also, multiple data buffers can be allocated
5 in PE memory so that different chunks of data can be input, processed and streamed out simultaneously.

Swizzle

Finally, the PEs are able to communicate with one another via what is known as the *swizzle* path that connects the register file of each PE with the register files of its left and right neighbours. On
10 each cycle, PE_n can perform a register-to-register transfer of 16 bits to either its left or right neighbour, PE_{n-1} or PE_{n+1} , while simultaneously receiving data from the other neighbour.

Swizzle instructions use multiples of 2 bytes in their arguments, so the source and destination registers must be 2-byte aligned. Instructions are provided to shift data left or right through the array, and to swap data between adjacent PEs.

15 The enable state of a PE affects its participation in a swizzle operation in the following way: if a PE is enabled, then its registers may be updated by a neighbouring PE, regardless of the neighbouring PE's enable state. Conversely, if a PE is disabled, its register file will not be altered by a neighbour under any circumstance. A disabled PE will still provide data to an enabled neighbour.

20 The data written into the registers of the PEs at the ends of the swizzle path can be set by the mono execution unit.

Host interface

The interfaces to the host system are used for 3 basic purposes: initialization and booting, access to host services and debugging.

25 Initialization

There are a number of stages of initialization required to start code running on the MTAP processor. These are normally handled transparently by the development tools, but an overview is provided here as background information.

First, the application code (including bootstrap) is loaded into memory.

30 Next, the host system initializes the state of the control unit, caches and mono execution unit by a series of writes to the PSCI port. The last of these specify the start address of the code to execute and tell the processor to start fetching instructions.

Finally, the boot code does any remaining initialization of the processor including any set-up of the PEs (e.g. setting the PE number) before running the application code.

Host services

Once application program is running it will need to access host resources such as the file system. A protocol is defined between the run-time libraries and a device driver on the host system. This is interrupt based and allows the code running on the MTAP processor to make calls to the application or operating system running on the host.

Debugging

The processor includes hardware support for breakpoints and single-stepping. These are controlled via registers accessed through the PCVI interface.

This interface also allows the debugger, running on the host, to interrogate and update the state of the processor to support fully interactive debugging.

Case Study: Device EV1

To illustrate the use of an MTAP processor in a real device, the architecture of Applicant's evaluation chip EV1 is described in this section. This device is a simple embodiment of Applicant's MTAP processor using Applicant's bus to interface the various ports of the processor to a memory block and to external pins.

EV1 architecture

Figure 9 shows the top level architecture of the EV1 device. The five ports of the MTAP processor and one port of the embedded memory block are connected to a single ClearConnect channel (ClearConnect is Applicant's bus structure) of six nodes and two lanes in opposing directions. The two external ports at each end of the bus allow the ClearConnect bus to be connected from one chip to another. This means a system can be built from multiple EV1 devices to provide the required performance.

The bus ports can also be used to connect to an FPGA to provide other functions, such as peripherals, an external memory controller or a host interface.

The EV1 chip also includes on-chip SRAM which provides fast access to code and data.

MTAP processor

The MTAP processor core in the EV1 has the following specification:

- General
 - Implemented on 0.13 μ process
 - Clock speed: 200 MHz
 - 4 Kbyte instruction cache: 4-way, 256 lines x 4 instructions, with manual and auto pre-fetch
 - 4 Kbyte data cache, 4-way, 256 lines x 16 bytes
- Mono execution unit

-19-

- 64 byte register file
- Support for 8 threads
- Poly execution unit
 - Array of 48 PEs
- 5 • MAC extension to the ALU
 - 4 Kbytes SRAM per PE
 - 64 byte register file
- One PIO channel
 - AVCI port: 32-bit address, 64-bit data
- 10 • Transfer size: 4, 8, 16, 32 bytes per PE
 - Address modes: direct and strided
- One SIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: up to 128 bytes per PE
- 15 • Control interfaces
 - Interrupt port: 32-bit address, 32-bit data PVCI
 - Register port: 32-bit address, 32-bit data PVCI

This gives a performance of:

- 9,600 MIPS
- 20 • 9.6 billion 8x8 MACs / second
- 38 Gbytes/s memory bandwidth
- 18 Gbytes/s inter-PE bandwidth

Example Applications

Here we give a couple of examples of how the MTAP processor can be applied to specific applications.

Network processing

In this case, the data to be processed consists of a stream of *packets* of variable size. Each packet consists of header information and a data payload. In its simplest form, the problem is to examine various fields in the header, do some sort of lookup function and route the packet to the appropriate output port. This processing must be done in real time.

30 In the network processing application, the SIO channels are used for continuously streaming packets into, and out of, PE memory. Within each PE, the software maintains several buffers so that, while one set of packets is being processed, the previous set can be output and, simultaneously, the next set can be loaded.

The PIO channels are used by the PEs to send requests to another subsystem that handles lookups. This could on-chip, for example Applicant's Table Lookup Engine, or an off-chip solution such as CAMs.

For more details see Emphasisparatextefault ¶ Font\11 sors[6].

5 **Bio-informatics**

The developing field of *in-silico* drug discovery uses computer simulation rather than the 'wet science' of test tubes to explore the behaviour of potential new drugs.

This includes molecular simulation of individual molecules and the interactions of molecules. For example, the docking of a ligand molecule (drug) into a protein. This simulation requires the calculation of the interaction energies of all atoms of one molecule with all the atoms of the other. This process is repeated for thousands of configurations of the molecules. This results in an embarrassingly parallel problem ideally suited to the MTAP architecture.

Each PE is allocated a different configuration of protein and ligand, and performs all of the atom-atom interaction energy calculations. The memory available within a PE is insufficient to hold the details of an entire molecule, however the molecule can be split into pieces and each piece processed in turn. The fetching of atoms can be overlapped with the atom-atom processing and, even with the floating point accelerated MTAP processor, is compute bound. The processing currently performed in most production code is a simplistic model of the interaction of atoms. The processing power that the MTAP architecture makes available allows more sophisticated models to be considered, improving the accuracy and thus value of the results.

The MTAP processor is also suited to other bio-informatic tasks such as genetic sequence comparison and pharmacophore fingerprinting.

Software Development Kit

The Software Development Kit (SDK) provides a complete set of development tools: C compiler, assembler linker, debugger, profiler, etc. The software tools use a central configuration file to define the attributes of the target processor that code is being generated for.

Simulation tools

Cycle and bit-accurate simulations of the core are available in C and Verilog. These can be parameterized for the number of PEs, size of memory, and other options. A generic configuration of the C simulator is shipped with the SDK. Once the target system is defined, the simulator can be configured to match that specification.

This allows application development to start before the system architecture is fully defined, and then proceed in parallel with the silicon implementation.

-21-

Once the target hardware (or a simulation) is available then the application code can be run and debugged on the target hardware.

Debugger

5 A major challenge when developing code for a parallel architecture is understanding the state of the machine when the code is stopped at a breakpoint or error. The data-parallel nature of the MTAP processor minimizes the difficulty here: a single instruction stream is executing and only the data is different on each PE.

10 Applicant's 2nd generation debugger builds on previous experience and provides a variety of ways to visualize the state of PE memory and registers. As well as traditional source-level symbolic views of data, and low-level dumps of memory and data, the debugger provides user-defined *picture* views which show the contents of PE memory or registers in graphical form. This provides a quickly understood view of the overall state of the system.

The debugger works identically with all of the simulators and with target hardware. It supports all the features expected from a modern development tool: graphical user interface and command
15 line operation, breakpoints, watchpoints, single-stepping, etc.



1 / 6

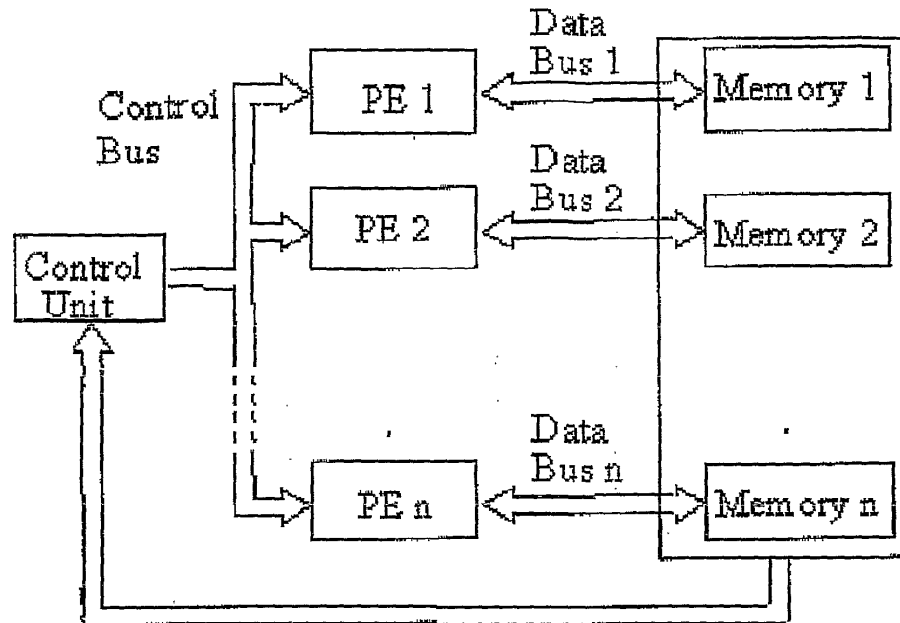


Fig 1

0082668 13-Oct-03 04:25





3 / 6

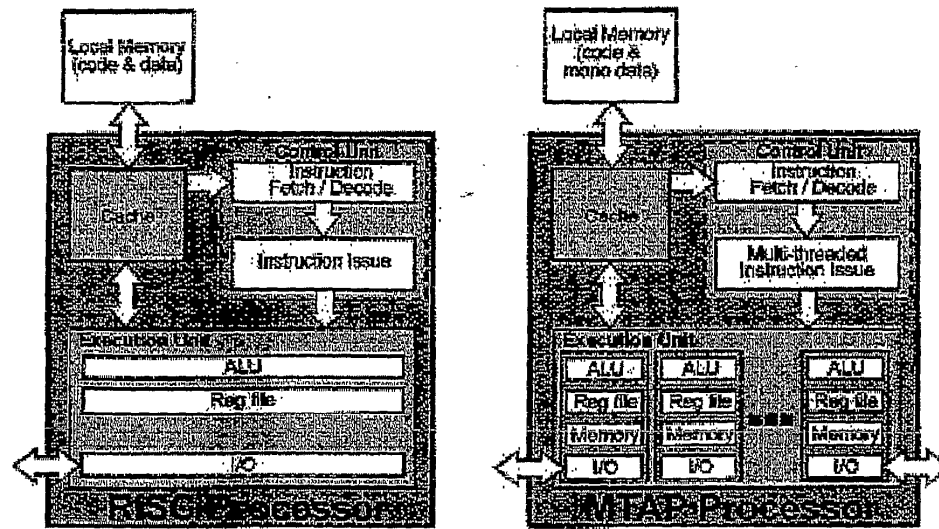


Fig 3

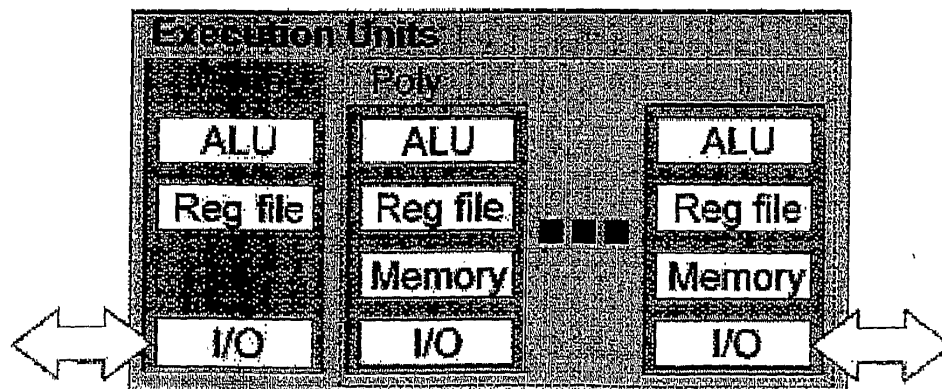


Fig 4

0082668 13-oct-03 04:25



4 / 6

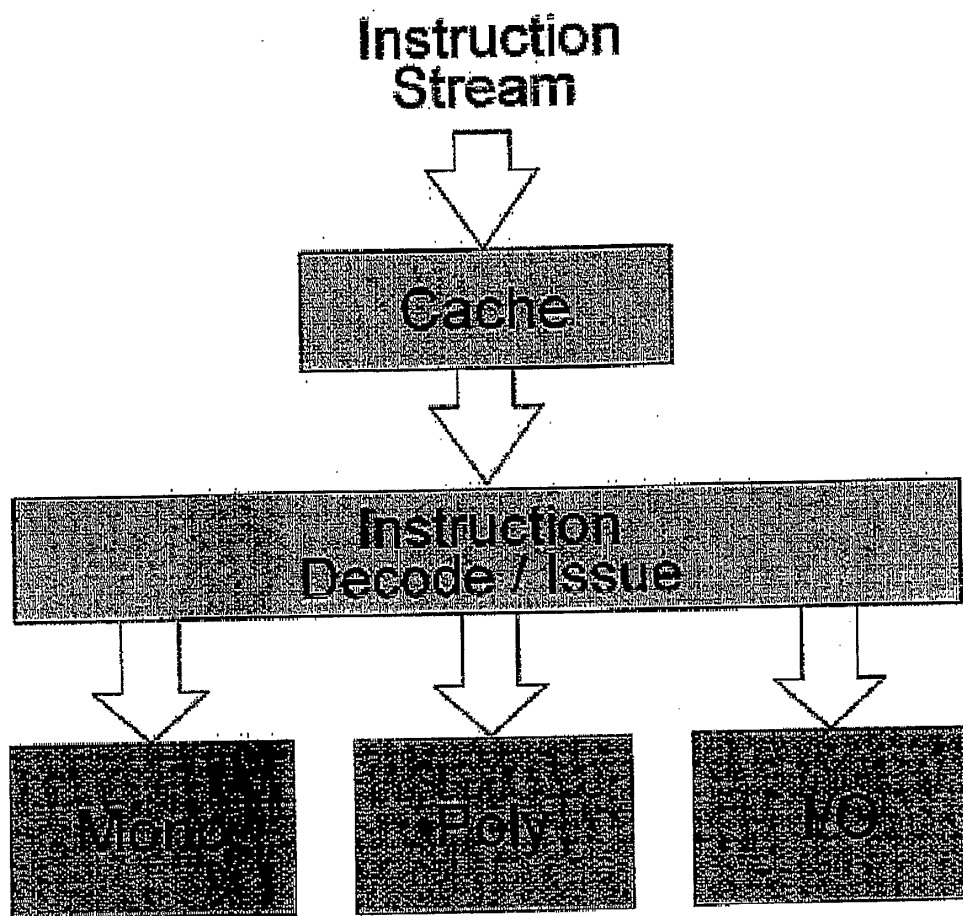
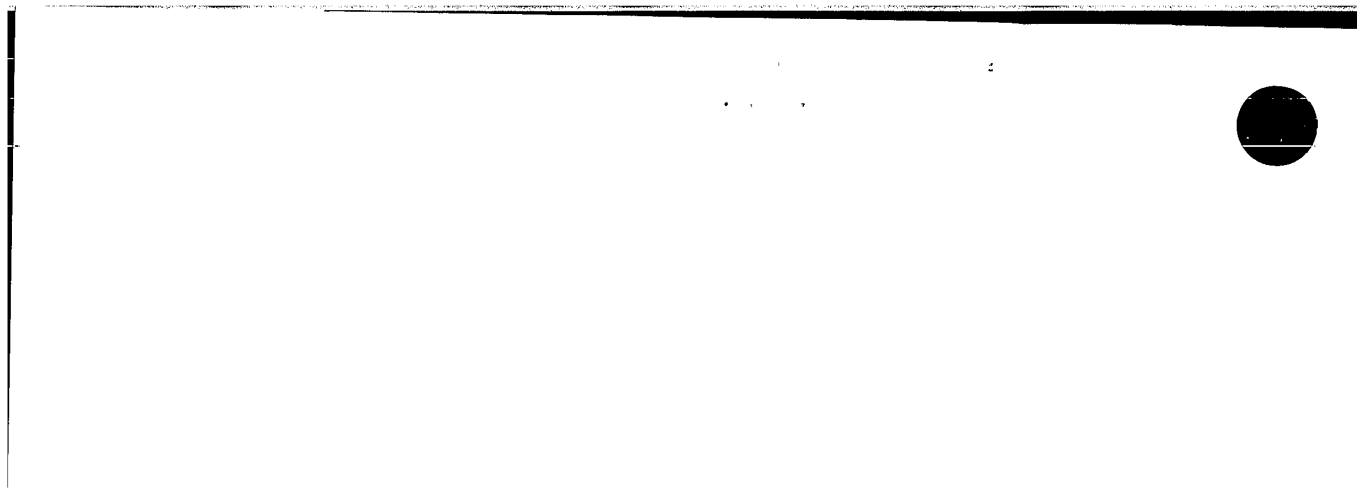


Fig 5



5 / 6

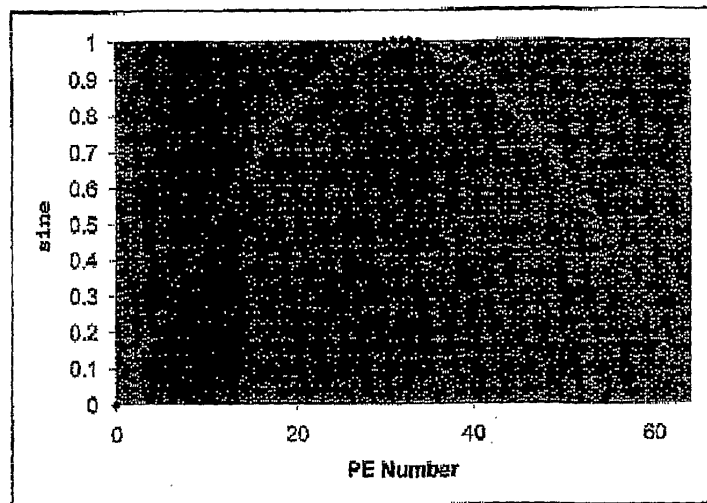


Fig 6

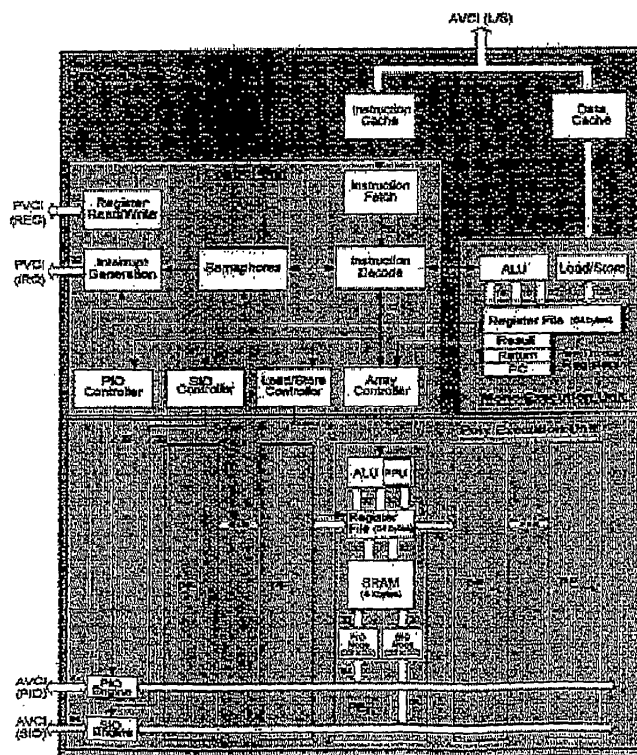


Fig 7



1000



6/6

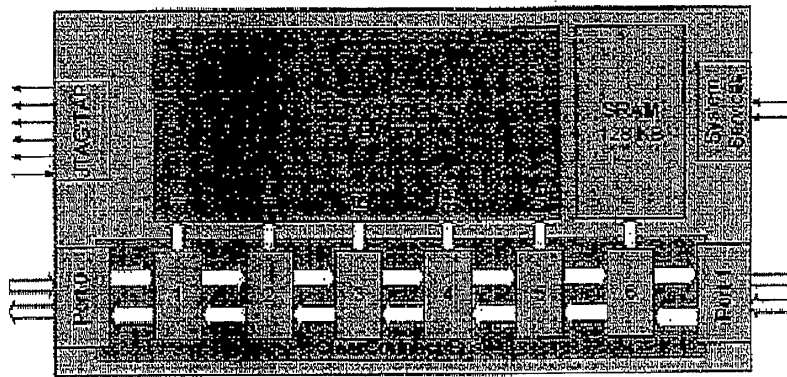


Fig 8





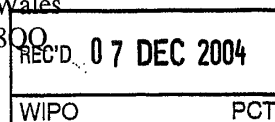
GB04/04377



INVESTOR IN PEOPLE

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN
COMPLIANCE WITH
RULE 17.1(a) OR (b)

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ



I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

I also certify that the attached copy of the request for grant of a Patent (Form 1/77) bears an amendment, effected by this office, following a request by the applicant and agreed to by the Comptroller-General.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed

Dated

22 October 2004



Patents Form 1/77

Patents Act 1977
(16)



1/77

Request for grant of a patent

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form)



The Patent Office

Cardiff Road
Newport
Gwent NP9 1RH

1. Your reference

P100390GB01/CIV

04MAY04 E892991-3 D02847

P01/7700 0.00-0409815.8 ACCOUNT CHA

2. Patent application number
(The Patent Office will fill in this part)

0409815.8

3. Full name, address and postcode of the or of each applicant (underline all surnames)

ClearSpeed Technology Limited
3110 Great Western Court
Hunts Ground Road
Stoke Gifford
Bristol BS34 8HP

08484933001

Patents ADP number (if you know it)

If the applicant is a corporate body, give the country/state of its incorporation

UNITED KINGDOM (See A/L 25 JUNE 2004)
KF 13 JULY 2004

4. Title of the invention
Unified SIMD Processor

5. Full name of your agent (if you have one)

Haseltine Lake

"Address for service" in the United Kingdom to which all correspondence should be sent (including the postcode)

Imperial House
15-19 Kingsway
London WC2B 6UD

Patents ADP number (if you know it)

34001

6. If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and (if you know it) the or each application number

Country

Priority application number
(if you know it)

Date of filing (day/month/year)

United Kingdom

0323950.6

13 October 2003

7. If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application

Number of earlier application

Date of filing (day/month/year)

8. Is a statement of inventorship and of right to a grant of patent required in support of this request? (Answer "Yes" if:
a) any applicant named in part 3 is not an inventor, or
b) there is an inventor who is not named as an applicant, or
c) any named applicant is a corporate body.
See note (d))

Yes

Patents Form 1/77

Patent Form 1/77

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document.

Continuation sheets of this form

Description	25
Claims(s)	4
Abstract	1
Drawing(s)	6

10. If you are also filing any of the following, state how many against each item.

Priority documents

Translations of priority documents

Statement of inventorship and right to a grant of patent (Patents Form 7/77)

Request for preliminary examination and search (Patents Form 9/77)

Request for substantive examination (Patents Form 10/77)

Any other documents (please specify)

Patents Form 23/77

11.

I/We request the grant of a patent on the basis of this application.

Haseltine Lake, Agents for the Applicants

Signature

Date

Haseltine Lake

28 April 2004

12. Name and daytime telephone number of person to contact in the United Kingdom

Chris Vigars

Tel: 0117 910 3200

Fax: 0117 910 3201

Warning

After an application for a patent has been filed, the Comptroller of the Patent Office will consider whether publication or communication of the invention should be prohibited or restricted under Section 22 of the Patents Act 1977. You will be informed if it is necessary to prohibit or restrict your invention in this way. Furthermore, if you live in the United Kingdom, Section 23 of the Patents Act 1977 stops you from applying for a patent abroad without first getting written permission from the Patent Office unless an application has been filed at least 6 weeks beforehand in the United Kingdom for a patent for the same invention and either no direction prohibiting publication or communication has been given, or any such direction has been revoked.

Notes

If you need help to fill in this form, please contact the Patent Office on 0117 910 3200.

DUPLICATE

Unified SIMD Processor

Background to the Invention

The invention consists of a significant improvement to a well known type of parallel processor. Single-Instruction Multiple-Data (SIMD) architectures are widely used in a number of computing applications. They have the advantage of high performance, low power consumption and easy scalability. A SIMD processor differs from the more commonly used multi-processor system (where a number of processors, each independently executing their own program code, are used together to solve a problem) in that it executes a single stream of instructions but each instruction operates on multiple data items in parallel.

There are a range of SIMD processor architectures. These range from an *intelligent memory* type of approach where each *column* of a memory array has a simple 1-bit Arithmetic Logic Unit (ALU) through to more complex architectures where a, usually smaller, array of more complex processing elements are fed with a single instruction stream. They have, however, been difficult to program in the past.

In all these cases the array of SIMD processing elements are fed with instructions and data by a dedicated controller. Often the controller is the largest and most complex part of the design because of the need to match the data processing rate achieved by the SIMD array. The array controller is not itself 'intelligent' and is, in turn, controlled by a general purpose processor which also runs the bulk of the application software. This *host* processor will execute most of the application code; it will send data to be processed to the SIMD array with an indication of the function to be performed, typically a pointer to the start of a sequence of SIMD array instructions or microinstructions. An exemplary prior art SIMD processor is outlined in Figure 1.

This introduces two programming problems. Firstly, the array controller can only be programmed at a very low level: a series of microcode, or possibly assembler, instructions to be executed by the array. This is because the array controller is not designed to work with a compiler. Secondly, this programming is completely separated from the main application code running on the host processor. This may be 'wrapped up' by providing C functions or C++ classes which makes them appear more integrated and reduces the complexity of passing control between the host processor and the SIMD array. Also, in the simplest form, the host processor and the SIMD array will have completely separate memory spaces for code and data. Any of

the standard shared memory techniques can be used to address this with all the resulting problems: complex arbitration, cache coherency issues, contention for memory bandwidth, inter-processor synchronization, greater cost, etc.

5 A preferable solution is to provide a fully integrated architecture where the same processor can transparently process both *scalar* and SIMD operations in a fully integrated fashion. This means that a single program can be written, in a standard high-level language such as C, which includes all parts of the application – both ‘normal’ (scalar) data and parallel data processing.

Prior Art

10 The basic SIMD processing model has been in use for many years. Some of the earliest examples are the ILLIAC IV developed in 1972, which had 64 processors with floating point and memory, and Goodyear’s STARAN (1975) which was a 1-bit architecture. Many variants have been developed since then.

15 There is also a recent trend to put small-scale SIMD extensions in standard processor architectures. This is exemplified by the Intel MMX and SSE extensions to the x86 architecture. This allows, for example, the values in two 32-bit registers to be added as if each register contained four independent 8-bit values. This is not considered relevant to the current discussion for a number of reasons:

- the SIMD processing is done on a very small scale;
- 20 • only a limited range of SIMD functionality is provided;
- there are distinct instructions for operating on SIMD data;
- the same hardware is used for SIMD and for scalar operations (although separate registers may be provided for the SIMD operands)

25 The idea of providing indexed addressing modes in a SIMD processor is disclosed in our copending patent application GB9908225.7. This invention extends that idea by using a more flexible implementation of that idea in combination with a more powerful array controller to provide a unified processor architecture.

Summary of the Invention

~~Applicant's approach after many years of experience with developing systems based~~

possible, identical – this makes it practical to write a compiler to target the processor. This greatly simplifies the task of programming the processor.

This means a single program runs on the processor which does both mono and poly operations as required – typically it will do both at the same time. A single C program
5 can be written which manages program control flow, operates on mono data, and operates on poly data.

To support this, and further simplify programming, this unity of the architecture is reflected in the instruction set and programming model. The processing elements in the SIMD array and the mono execution unit have a common instruction set. For
10 instance, an **add** instruction can be used with operands which are either poly registers or mono registers (or immediate values); the appropriate execution unit will execute the instruction. Mono and poly operands can also be mixed to allow, for example, a single mono value to be added to a poly register which holds a different value on every processing element.

15 It is appropriate to note that this does not require the mono execution unit to have exactly the same micro-architecture as the poly execution unit. For example, the poly processing elements could have an 8-bit ALU, while the mono execution unit has a 32-bit ALU.

A block schematic diagram of a processor in accordance with the invention is shown
20 in Figure 2.

The most significant features contributing to the novelty and inventiveness of this invention

include the following concepts:

- The processor provides a unified programming model: a single program runs on
25 a processor providing both scalar and SIMD operations.
- This is supported with a unified instruction set: e.g. an **add** instruction can work with either mono or poly operands (or a mixture of the two)
- Poly operations can freely use mono registers in expressions, and more importantly for addressing memory.
- 30 • Having a full range of addressing modes and completely separate memory blocks in each PE make it simple to support poly data structures and poly pointers on the SIMD array (previous SIMD architectures either provide a single address to all PEs or use sequences of instructions to construct an address in memory).

- Mono and poly operations are of equal standing: wherever possible, all instructions can be used with either mono or poly operands (there are some exceptions because a small set of operations, mainly for control, only make sense for one or the other).
- 5 • The mono and poly execution units are loosely coupled so that their operation can be overlapped. Compilers are extremely good at scheduling instructions to take advantage of this flexibility - maximizing the efficiency of the system.
- There are some compiler optimizations specific to this architecture: for example it may be better, in some instances, to move an operation that would normally
10 be done on the mono execution unit on to the poly execution unit - either because it can then be overlapped with other operations on the mono execution unit or because the result is needed on the poly execution unit anyway. Again, the compiler can detect such cases and make the appropriate optimizations.
- A common code and data space is available to both mono and poly operations
15 making it easy to generate code for, and share data between, the two execution units.
- The overall architecture is suitable for targeting by a high level language compiler - again for both mono and poly operations.

20 To this end, the invention provides, in a first aspect a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set.

According to a second aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel
25 and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data.

According to a third aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel

comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.

5 The single control unit and the integral execution unit may be adapted to operate asynchronously on both the parallel and scalar data under the same instruction set. The execution unit preferably comprises a SIMD part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data.

10 Preferably, the SIMD part and the scalar part differ only in that each processing element of the array contains local memory. The SIMD part and the scalar part preferably have a common memory area for instructions and scalar data. Each PE may contain an ALU and one or more multiply-accumulate function units or an ALU and one or more floating point function units. Preferably, each PE has multiple
15 enable bits to support nested conditional code.

The SIMD and the scalar execution unit conveniently have a similar set of status/result flags which can be used to control branching on the scalar part of the execution unit and conditional execution on the SIMD part of the execution unit in a similar manner.

20 There is preferably a communication path between PEs in the SIMD part and means are preferably provided for data to be transferred to and from the scalar part of the execution unit and the SIMD part of the execution unit.

Advantageously the control unit is adapted to fetch and execute multiple instruction threads simultaneously. A semaphore unit may use semaphores to synchronize
25 between instruction threads. The semaphores may be used to synchronize between I/O operations and instruction threads. The control unit may be adapted to schedule and execute threads based on their priority and the state of any semaphores they are waiting on and may include a set of the semaphores for use in synchronizing threads. Each processing element of the SIMD array may comprise one or more function units,
30 a register file, said local memory, and I/O means.

Both the scalar part and the SIMD part preferably have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the

execution unit; SIMD variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory.

Substantially all the instructions are preferably operable on either the parallel data or the scalar data or a combination of both the scalar data and the parallel data.

- 5 The processor may further comprise a compiler adapted to produce code from a single source program to operate on both the scalar and the parallel data. The compiler is preferably adapted to detect parallel data automatically.

The scalar part, the SIMD part, and the I/O means are preferably adapted to be operated in parallel by the same instruction stream.

- 10 A programmer or the compiler is adapted to use the multi-threaded execution to schedule I/O operations such that they occur concurrently with computation and data is always available when required.

The I/O means may comprise programmed I/O means adapted to enable each PE to transfer data to and from external scalar memory and wherein each PE provides the

- 15 address it wants to read data from or write data to. The memory accesses may be consolidated to minimize the number of memory transfers when multiple PEs are accessing the same memory area.

Each PE is preferably adapted to transfer data to and from external scalar memory, in which case the address for each PE is generated automatically based on the PE

- 20 number and the amount of data being transferred.

The I/O means may comprise streaming I/O means which distributes an incoming data stream to all the PEs in the SIMD part and collects data from all the PEs to generate an output stream independently of program execution on the SIMD part of the execution unit. The streaming I/O means may only use enabled PEs to take part in

25 the streaming I/O operation. The size of data distributed to, and collected from each PE may be different for each PE.

Brief Description of the Drawings

The invention will be described with reference to the following drawings, in which:

Figure 1 shows a prior art SIMD processor.

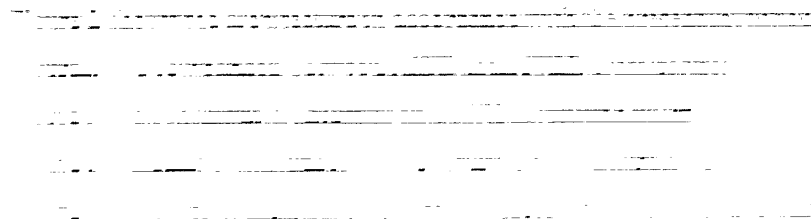


Figure 7 is a block schematic diagram depicting an exemplary MTAP processor; and

Figure 8 is a block diagram of a specific implementation of the MTAP architecture as applied to an evaluation chip.

5 Detailed Description of the Illustrated Embodiments

The design of this processor architecture was driven by the desire to make it practical and efficient to program in a high level language. In order to make this possible a number of features are required. First, the overall programming model needs to be simple and 'regular' - i.e. the same operations and addressing modes can be applied to
10 all data types.

The execution unit for mono data is based on well known principles used in many RISC processors. This makes it simple for a compiler to target and easy for a user to understand.

The functionality supported by each PE is then made as similar as possible to the
15 mono execution unit: each PE has an ALU, a register file and memory, and supports a range of addressing modes for transferring data between memory and registers.

It is not necessary for the PEs to be identical to the mono execution unit in every detail: e.g. the number of registers and the width and complexity of the ALU are likely to be different in practice. The poly execution unit gets its performance largely
20 from the number of PEs brought to bear on the problem rather than the processing power of the individual PEs. To balance this, the mono execution unit is likely to require a wider and more complex ALU. This is reasonable because this is only instantiated once while the PEs are replicated many times and need to be efficiently implemented.

25 The number of clock cycles required to execute an instruction on the mono and poly execution units is highly variable. Both will execute some instructions in a single cycle. But some instructions could take several cycles on the mono execution unit or on the poly execution unit. It is therefore essential that the two execution units are only loosely coupled rather than operating in lock-step; this allows them to overlap
30 their execution of instructions. The hardware will synchronize their operations when necessary: e.g. when they need to access a shared resource, or one execution unit requires data from the other.

Transfers between the registers and memory, and other I/O transfers, can take place concurrently with ALU operations. To simplify the task for the compiler, these

interactions are all interlocked using register scoreboarding (see our copending patent application GB 9908203.4).

An important aspect of the PE architecture to support compiled code is the existence of a full set of addressing modes: direct, indirect and indexed. Indirect and indexed
5 addressing by the PEs is described in the previously mentioned copending patent application GB 9908225.7.

Reference may also be had to our copending patent application GB 0321186.9 for details of the "ClearConnect" bus referred to in the context of the present invention.

In a high level language compiler for this architecture, it is only necessary to
10 introduce a simple modification; the keyword 'poly' used in a declaration identifies data that is distributed across the SIMD array. Similar methods have been used before, for example the 'shape' keyword used in C*, a language used for programming the "Connection Machine". However, it is only with the present architecture that poly variables become 'first class' objects - i.e. they can be used everywhere that 'normal'
15 mono data can be: in expressions, as function arguments, in conditional statements, etc. Pointers to both mono and poly data are allowed, as are pointers of both mono and poly types. Furthermore, poly variables can be mixed quite freely with mono data in all these contexts. This provides a very simple and regular programming model for the user.

20 Note that it would also be possible for a compiler to detect the parallelism in the program or data automatically: This is frequently done with 'vectorizing' Fortran compilers used with supercomputers, for example. The same techniques could be applied to this architecture. The architecture will now be described in more detail.

The Multi-Threaded Array Processing (MTAP) architecture has been developed to
25 address a number of problems in high performance, high data rate processing.

A number of applications are characterized by very high data rates, flexible processing requirements and hard real-time constraints. We use the term *data flow processing* to describe this class of problem.

~~The MTAP processor delivers on the three primary requirements for data-flow~~

2. It provides the raw horsepower for the processing functions required, on the maximum data rate that the system will encounter. That processing power can scale with increasing function demands.
3. It stores the data close to the processing core to maximize bandwidth and minimize latency.

MTAP Overview

The MTAP architecture defines a family of embedded processors with parallel data processing capability. Figure 3 compares a standard processor and the MTAP architecture. As can be seen, the MTAP processor has a standard, RISC-like, control unit with instruction fetch, caches and I/O mechanisms. This is coupled to a highly parallel execution unit which provides the performance and scalability of the MTAP architecture.

The processor is designed in a highly modular fashion which allows many details of a specific implementation to be easily customized for the target application.

- 15 To simplify the integration of the processor into a variety of systems, the processor can be also configured to be big or little-endian.

Control unit

The control unit fetches, decodes and dispatches instructions to the execution units. The processor executes a fairly standard, three operand instruction set. The manner in which the same instruction stream is decoded and issued in parallel to the mono part, the poly part and the I/O part is schematically indicated in Figure 5.

The control unit also provides hardware support for multi-threaded execution, allowing fast swapping between multiple threads. The threads are prioritized and are intended primarily to support efficient overlap of I/O and compute. This can be used to hide the latency of external data accesses.

The MTAP processor can also include instruction and data caches to minimize the latency of memory accesses. The size and type of these is a configurable option. Alternatively, the caches may be replaced with SRAM to provide a complete embedded solution.

- 30 The control unit also includes a control port which is used for initializing and debugging the processor; it includes support for breakpoints, single stepping and the examination of internal state.

Execution unit

The execution unit consists of a number of *processing elements* (PEs). This allows it to process data elements in parallel. Each PE consists of an ALU, registers, memory and I/O.

- 5 The number of PEs in a processor core is a configurable parameter which allows performance to be scaled to meet the needs of the application (see Figure 6). The execution unit can be thought of as two, largely independent, parts (see Figure 4).

One PE forms the *mono* execution unit; this is dedicated to processing mono (i.e. scalar or non-parallel) data. The mono execution unit also handles program flow control such as branching and thread switching. The rest of the PEs form the *poly* execution unit which processes parallel (poly) data.

The poly execution unit may consist of tens, hundreds or even thousands of PEs. This array of PEs operates in a synchronous manner, similar to SIMD, where every PE executes the same instruction on its piece of data. Each PE also has its own independent local memory; this provides fast access to the data being processed. For example, one PE at 400 MHz has a memory bandwidth of 1.6 Gbytes/s. An array with 256 such PEs has an aggregate bandwidth of over 400 Gbytes/s, with single cycle latency. The number of registers and the amount of memory in each PE are configurable options.

Input / output

The MTAP processor core has two basic I/O mechanisms. The first, *Programmed I/O* (PIO), is the normal mechanism used for accessing memory external to the MTAP core: it supports random accesses to variable sized data by the PEs. The second mechanism is *Streaming I/O* (SIO) which allows chunks of contiguous data to be streamed directly into the memory within PEs.

Each of these supports a variety of addressing modes, described in more detail later.

Programming model

From a programmer's perspective, the MTAP processor appears as a single processor running a single C program. This is very different from some other parallel processing

I/O controllers. The processor can despatch an instruction on every cycle.

For multi-cycle instructions, the operation of the functional units can be overlapped. So, for example, an I/O operation can be started with one instruction and on the next cycle the mono execution unit could start a multiply instruction (which requires several cycles to execute). While these operations are proceeding, the poly execution unit can continue to execute instructions.

The main change from programming a standard processor is the concept of operating on parallel data. Data to be processed is assigned to variables which have an instance on every PE and are operated on in parallel: we call this *poly* data. This can be thought of as a data vector which is distributed across the array.

Variables only requiring a single instance (e.g. loop control variables) are known as *mono* variables – they behave exactly like normal variables on a sequential processor. Applicant provides a compiler which uses a simple extension to standard C to identify data which is to be processed in parallel. The new keyword **poly** is used in a declaration to define data which exists, and is processed, on every PE in the poly execution unit.

Example

To give a feel for the way the MTAP processor is programmed, the following represents a fragment of code which will calculate 64 values of a sine function across the PE array in a single operation.

```
#include <fnext.h>
#include <math.h>
#define PI 3.1415926535897932384
#define NUMBER_OF_PES 64

poly float angle, sine;
poly int pe;

/* get PE number: 0...n-1 */
pe = get_penum();
/* convert to an angle in range 0 to Pi */
angle = pe * PI / NUMBER_OF_PES;
/* calculate sine of angle on each PE */
sine = sinp(angle);
```

This code uses the library function `get_penum()` to get a unique value in the variable `pe` on each PE. This is scaled to give a range of values for `angle` between 0 and π across the PEs.

Finally, the library function `sinp()` is called to calculate the sine of these values on all PEs simultaneously. The `sinp` function is the poly equivalent of the standard sine function; it takes a poly argument and returns the appropriate value on every PE.

Performance

The performance of an MTAP core with 64 PEs, on a 0.13μ process and running at 400 MHz is shown in Table 1. The columns show the performance achieved with the basic PE, the PE with an additional Multiply-Accumulate (MAC) unit, and the PE with an FPU extension.

Table 1 - Performance Comparison

	Base Architecture	8-bit MAC	FPU
MIPS	25,600	25,600	25,600
MMACS ($\times 10^6$)	—	25,600	25,600
MFLOPS	—	—	51,200
Memory Bandwidth (Gbytes / s)	100	100	100
Inter-PE Bandwidth (GBytes / s)	50	50	50
Relative Area	1.0	1.1	1.3

However, the core can be scaled well beyond this number of PEs, to 256 or even thousands of PEs. It also scales down efficiently to lower numbers of PEs, or lower clock rates, enabling low cost and low power systems to be built.

Figure 6 shows how performance and power scale with the number of PEs. By scaling the number of PEs and the clock frequency, the core can be scaled to meet the requirements of a wide range of applications.

Architecture Details

The following sections describe the MTAP processor in more detail. A block diagram of the processor is shown in Figure 7. This is a conceptual view, which reflects the programming model. In practice, the control and mono execution units form a tightly-coupled pipeline which provides overall control of the processor.

Interfaces

The processor has a number of external interfaces. The widths of each of these are configurable. All these interfaces currently use the Virtual Component Interface Standard defined by the VSI Alliance.

There are three categories of interface:

Mono data & instructions

This is a single *Advanced VCI* (AVCI) interface which is used for mono loads and stores, and for instruction fetching.

Poly data

This uses one or more AVCI interfaces for PIO and SIO data. The number of physical interfaces corresponds to the number of PIO and SIO channels implemented.

Control

There are two control interfaces which use the *Peripheral VCI* (PVCi) protocol. One is used for initialization and debug. The other allows the MTAP processor to generate interrupts to the host system.

Instruction set

The processor has a fairly standard RISC-like instruction set. Most instructions can operate on mono or poly operands and are executed by the appropriate part of the execution unit. Some instructions are only relevant to either the mono or poly execution unit, for example all program flow control is handled by the mono unit. The instruction set provides a standard set of functions on both mono and poly execution units:

- Integer and floating point adds, subtracts, multiplies, divides
- Logical operations: and, or, not, xor
- Arithmetic and logical shifts
- Data comparisons: equal, not equal, greater than, less than, etc.
- Data movement between registers
- Loads and stores between registers and memory

To give a feel for the nature of assembly code for the MTAP processor, a few lines of code are shown below. This simple example loads a value from memory into a mono register, gets the PE number into a register on each PE and then adds these two values together producing a different result on every PE.

```
5  ld 0:m4, 0x3000    // load mono reg 0 from mem
   penum 8:p4         // PE number into poly reg
   add 4:p4, 0:m4, 8:m4 // add; result in reg 4
```

Control unit

The control unit fetches instructions from memory, decodes them and despatches them to the appropriate functional unit.

The controller includes a scheduler to provide hardware support for multi-threaded code. This is a vital part of the architecture for achieving the performance potential of the MTAP processor. Because of the highly parallel architecture of the poly execution unit, there can be significant latencies if all PEs need to read or write external data.

15 When part of an application stalls because it is waiting for data from external memory, the processor can switch to another code thread that is ready to run. This serves to hide the latency of accesses and keep the processor busy.

The number of threads is a configurable option. The threads are prioritized: the processor will run the highest priority thread that is ready to run; a higher priority thread can pre-empt a lower priority thread when it becomes ready to run. Threads are synchronized – with each other and with hardware, such as I/O engines – via semaphores.

In the simplest case of multi-threaded code, a program would have two threads: one for I/O and one for compute. By *pre-fetching* data in the I/O thread, the programmer
25 (or the compiler) can ensure the data is available when it is required by the execution units and the processor can run without stalling.

Execution units

This section describes the common aspects of the poly and mono execution units.

ALU operations

Arithmetic and logical operations are provided in several variants:

Status register

Associated with the ALU is a status register; this contains five status bits that provide information about the result of the last ALU operation. When set, these bits indicate:

- Most significant bit set
- 5 • Carry generated
- Overflow generated
- Negative result
- Zero result

Registers

- 10 To support operations on data of different widths, the registers in the PEs can be accessed very flexibly. The register files are best thought of as an array of bytes which can be addressed as registers of 1 to 8 bytes wide.

- The mono and poly registers are addressed in a consistent way using byte addresses and widths specified in bytes. The mono register file is 16 bits wide and so all
- 15 addresses and widths must be a multiple of 2. There are no alignment restrictions on poly register accesses.

Addressing modes

- Load and store instructions are used to transfer data between memory and registers. In the case of the mono execution unit, these transfer data to and from memory external
- 20 to the MTAP processor.

Poly loads and stores transfer data between the PE register file and PE memory. Data is transferred between the PEs and external memory using the I/O functions described later.

- There are three addressing modes for loads and stores which can be used for both
- 25 mono and poly data. These are:

Direct

The address to be read/written is specified as an immediate value.

Indirect

The address is specified in a register.

- 30 **Indexed**

The address is calculated from adding an offset to a base address in a register. The offset must be an immediate value.

Conditional code

The main difference between code for the mono and the poly execution units is the handling of conditional execution.

The mono unit uses conditional jumps to branch around code, typically based on the result of the previous instructions. This means that mono conditions affect both mono and poly operations. This is just like a standard RISC architecture.

The poly unit uses a set of enable bits (described in more detail below) to control whether each PE will have its state changed instructions it executes. This provides per-PE predicated operation. The enable state can be changed based on the result of a previous operation.

The following sections describe the architectural features of the two execution units in more detail.

Mono execution unit

The mono execution unit is a 16-bit processing element consisting of:

- A 16-bit ALU with optional extensions
- A register file of configurable size
- Status and control registers

The mono ALU extensions include a multiplier, a barrel shifter and a normalizer, which is important for accelerating software implementations of floating point operations.

As well as handling mono data, the mono unit is responsible for program flow control (branching), thread switching and other control functions. The mono execution unit also has overall control of I/O operations of poly data. Results from these operations are returned to a register in the mono unit.

Conditional execution

The mono execution unit handles conditional execution in the same way as a traditional processor. A set of conditional and unconditional jump instructions use the result of previous operations to jump over conditional code, back to the start of loops, etc.

The threads are prioritized (0 being highest priority). Control of execution between the threads is performed using semaphores under programmer control. Higher priority threads will only yield to lower priority threads when they are stalled on yielding instructions (such as semaphore wait operations). Lower priority threads can be pre-empted at any time by higher priority ones.

Semaphores are special registers that can be incremented or decremented with atomic (non-interruptible) operations called *signal* and *wait*. A *signal* instruction will increment a semaphore. A *wait* will decrement a semaphore unless the semaphore is 0, in which case it will stall until the semaphore is signalled by another thread. Semaphores can also be accessed by hardware units (such as the I/O controllers) to synchronize these with software.

Poly execution unit

The poly execution unit is an array of Processing Elements (PEs). Each PE in the poly execution unit consists of:

- An 8-bit ALU with optional extensions such as a multiply-accumulate (MAC) unit
- A register file of configurable size
- Status and enable registers
- A block of memory of configurable size
- An inter-PE communication path
- One or more I/O channels

Load and store instructions move data between a PE's register file and memory, while the ALU operates on data in the register file. Data is transferred in to, and out of, the PE's memory using I/O instructions.

ALU

The poly ALU is used for performing arithmetic and logical operations on values held in the PE register file.

While the ALU is only 8 bits wide, instructions exist for multi-byte arithmetic which is handled by iteration. ALU extensions exist to accelerate functions for floating point, DSP, etc. For example, an optional integer multiply-accumulate unit (MAC) can be included in the ALU. This can deliver an 8 x 8 bit MAC result every cycle, or a 16 x 16 bit MAC every four cycles. The accumulated result can be up to 64 bits wide. The standard ALU includes basic hardware support to accelerate floating point operations.

Conditional behaviour

The SIMD nature of the PE array prohibits each PE having its own branch unit (branching being handled by the mono execution unit). Instead, each PE can control whether its state should be updated by the current instruction by *enabling* or *disabling* itself; this is rather like the predicated instructions in some RISC CPUs.

Enable state

A PE's enable state is determined by a number of bits in the enable register. If all these bits are set to one, then a PE is *enabled* and executes instructions normally. If one or more of the enable bits is zero, then the PE is *disabled* and most instructions it receives will be ignored (instructions on the enable state itself, for example, are not be disabled).

The enable register is treated as a stack, and new bits can be pushed onto the top of the stack allowing nested predicated execution. The result of a test, either a 1 or a 0, is pushed onto the enable stack. This bit can later be popped from the top of the stack to remove the effect of that condition. This makes handling nested conditions and loops very efficient. Note that, although the enable stack is of fixed size, the compiler handles saving and restoring the state automatically, so there are no limitations on compiled code. When programming at the assembler level, it is the programmer's responsibility to manage the stack.

Instructions

Conditional execution on the poly execution unit is supported by a set of poly conditional instructions: **if**, **else**, **endif**, etc. These manage the enable bits to allow different PEs to execute each branch of an **if...else** construct in C, for example. These also support nested conditions by pushing and popping the condition value on the enable stack.

As a simple example, consider the following code fragment:

```
// disable PEs where reg 32 is non-zero
if.eq 32:p1, 0 // push result onto stack
// increment reg 8 on enabled PEs
```

```
add     r8, r8, r1
```

were previously enabled and disabled if they were already disabled. If the two operands are not equal, a 0 is pushed onto the stack – this disables the corresponding PEs.

5 The following **add** instruction is sent to all PEs, but only acted on by those that are still enabled. Finally, the **endif** instruction pops the enable stack, returning all PEs to their original enable state.

Forced loads and stores

10 Poly loads and stores are normally predicated by the enable state of the PE. However, because there are instances where it is necessary to load and store data regardless of the current enable state, the instruction set includes *forced* loads and stores. These will change the state of the PE even if it is disabled.

I/O mechanisms

15 There are two I/O mechanisms provided for transferring data between PE memory and devices outside the MTAP core. *Programmed I/O* (PIO) extends the load/ store model: it is used for transfers of small amounts of data between PE memory and external memory. *Streaming I/O* (SIO) is used to efficiently transfer contiguous chunks of data in and out of PE memory.

20 The number of PIO and SIO channels in a processor is a configurable parameter. A processor will always have at least one PIO channel as this is required for normal program execution.

Multiple I/O channels can run simultaneously.

I/O architecture

The I/O systems consist of three parts: Controller, Engine and Node.

Controller

25 The PIO and SIO controllers decode I/O instructions and coordinate with the rest of the control unit and the mono processor. The controllers synchronize with software threads via semaphores.

Engine

The I/O engines are basically DMA engines which manage the actual data transfer.

30 There is a Controller and Engine for each I/O channel. A single Controller can manage several I/O Engines.

Node

There is an I/O Node in each PE. The I/O Engine activates each Node in turn allowing to serialize the data transfers. The Nodes provide buffering of data to minimize the impact of I/O on the performance of the PEs.

5 **Programmed I/O (PIO)**

PIO is closely coupled to program execution and is the normal way for the processor to transfer data to and from the outside world (e.g. external memory, hardware accelerators or a host processor).

The PIO mechanism provides a number of addressing modes:

10 **Direct addressed**

Each PE provides an external memory address for its data. This provides random access to data.

Strided

15 The external memory address is incremented for each PE. In each case, the size of data transferred to each PE is the same.

When multiple PEs are accessing memory then the transfers can be *consolidated* so as to perform the minimum number of external accesses. So, for example, if half the processors are reading one location and the other half reading another, then only two memory reads would be performed. In fact, consolidation can be better than that:
20 because the bus transfers are *packetized*, even transfers from nearby addresses can be effectively consolidated.

Streaming I/O (SIO)

SIO is used for streaming high-bandwidth data directly to and from PE memory. This is less flexible but very efficient for transferring blocks of data in and out of the
25 system. This is typically used to stream data to and from memory mapped I/O devices. Each PE can transfer a different size block of data.

Streaming I/O can be made very efficient in a number of ways. For example, multi-threaded code allows data I/O and compute to be fully overlapped. Also, multiple data buffers can be allocated in PE memory so that different chunks of data can be input.

16 bits to either its left or right neighbour, PE_{n-1} or PE_{n+1} , while simultaneously receiving data from the other neighbour.

Swazzle instructions use multiples of 2 bytes in their arguments, so the source and destination registers must be 2-byte aligned. Instructions are provided to shift data left or right through the array, and to swap data between adjacent PEs.

The enable state of a PE affects its participation in a swazzle operation in the following way: if a PE is enabled, then its registers may be updated by a neighbouring PE, regardless of the neighbouring PE's enable state. Conversely, if a PE is disabled, its register file will not be altered by a neighbour under any circumstance. A disabled

PE will still provide data to an enabled neighbour.

The data written into the registers of the PEs at the ends of the swazzle path can be set by the mono execution unit.

Host interface

The interfaces to the host system are used for 3 basic purposes: initialization and booting, access to host services and debugging.

Initialization

There are a number of stages of initialization required to start code running on the MTAP processor. These are normally handled transparently by the development tools, but an overview is provided here as background information.

First, the application code (including bootstrap) is loaded into memory.

Next, the host system initializes the state of the control unit, caches and mono execution unit by a series of writes to the PSCI port. The last of these specify the start address of the code to execute and tell the processor to start fetching instructions.

Finally, the boot code does any remaining initialization of the processor including any

set-up of the PEs (e.g. setting the PE number) before running the application code.

Host services

Once application program is running it will need to access host resources such as the file system.

A protocol is defined between the run-time libraries and a device driver on the host system. This is interrupt based and allows the code running on the MTAP processor to make calls to the application or operating system running on the host.

Debugging

The processor includes hardware support for breakpoints and single-stepping. These are controlled via registers accessed through the PSCI interface.

This interface also allows the debugger, running on the host, to interrogate and update the state of the processor to support fully interactive debugging.

Case Study: Device EV1

To illustrate the use of an MTAP processor in a real device, the architecture of Applicant's evaluation chip EV1 is described in this section. This device is a simple embodiment of Applicant's MTAP processor using Applicant's bus to interface the various ports of the processor to a memory block and to external pins.

EV1 architecture

Figure 9 shows the top level architecture of the EV1 device. The five ports of the MTAP processor and one port of the embedded memory block are connected to a single **ClearConnect** channel (**ClearConnect** is Applicant's bus structure) of six nodes and two lanes in opposing directions. The two external ports at each end of the bus allow the **ClearConnect** bus to be connected from one chip to another. This means a system can be built from multiple EV1 devices to provide the required performance.

The bus ports can also be used to connect to an FPGA to provide other functions, such as peripherals, an external memory controller or a host interface.

The EV1 chip also includes on-chip SRAM which provides fast access to code and data.

MTAP processor

The MTAP processor core in the EV1 has the following specification:

- General
 - Implemented on 0.13 μ process
 - Clock speed: 200 MHz
 - 4 Kbyte instruction cache: 4-way, 256 lines x 4 instructions, with manual and auto pre-fetch
 - 4 Kbyte data cache, 4-way, 256 lines x 16 bytes
- Mono execution unit
 - 64 byte register file

- 64 byte register file
- One PIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: 4, 8, 16, 32 bytes per PE
- 5 • Address modes: direct and strided
- One SIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: up to 128 bytes per PE
- Control interfaces
- 10 • Interrupt port: 32-bit address, 32-bit data PSCI
- Register port: 32-bit address, 32-bit data PSCI

This gives a performance of:

- 9,600 MIPS
- 9.6 billion 8x8 MACs / second
- 15 • 38 Gbytes/s memory bandwidth
- 18 Gbytes/s inter-PE bandwidth

Example Applications

Here we give a couple of examples of how the MTAP processor can be applied to specific applications.

20 ***Network processing***

In this case, the data to be processed consists of a stream of *packets* of variable size. Each packet consists of header information and a data payload. In its simplest form, the problem is to examine various fields in the header, do some sort of lookup function and route the packet to the appropriate output port. This processing must be
25 done in real time.

In the network processing application, the SIO channels are used for continuously streaming packets into, and out of, PE memory. Within each PE, the software maintains several buffers so that, while one set of packets is being processed, the previous set can be output and, simultaneously, the next set can be loaded.

30 The PIO channels are used by the PEs to send requests to another subsystem that handles lookups. This could on-chip, for example Applicant's Table Lookup Engine, or an off-chip solution such as CAMs.

Bio-informatics

The developing field of *in-silico* drug discovery uses computer simulation rather than the 'wet science' of test tubes to explore the behaviour of potential new drugs.

- This includes molecular simulation of individual molecules and the interactions of molecules. For example, the docking of a ligand molecule (drug) into a protein. This simulation requires the calculation of the interaction energies of all atoms of one molecule with all the atoms of the other. This process is repeated for thousands of configurations of the molecules. This results in an embarrassingly parallel problem ideally suited to the MTAP architecture.
- Each PE is allocated a different configuration of protein and ligand, and performs all of the atom-atom interaction energy calculations. The memory available within a PE is insufficient to hold the details of an entire molecule, however the molecule can be split into pieces and each piece processed in turn. The fetching of atoms can be overlapped with the atom-atom processing and, even with the floating point accelerated MTAP processor, is compute bound. The processing currently performed in most production code is a simplistic model of the interaction of atoms. The processing power that the MTAP architecture makes available allows more sophisticated models to be considered, improving the accuracy and thus value of the results.
- The MTAP processor is also suited to other bio-informatic tasks such as genetic sequence comparison and pharmacophore fingerprinting.

Software Development Kit

- The Software Development Kit (SDK) provides a complete set of development tools: C compiler, assembler linker, debugger, profiler, etc. The software tools use a central configuration file to define the attributes of the target processor that code is being generated for.

Simulation tools

Cycle and bit-accurate simulations of the core are available in C and Verilog. These can be parameterized for the number of PEs, size of memory, and other options. A

Once the target hardware (or a simulation) is available then the application code can be run and debugged on the target hardware.

Debugger

5 A major challenge when developing code for a parallel architecture is understanding the state of the machine when the code is stopped at a breakpoint or error. The data-parallel nature of the MTAP processor minimizes the difficulty here: a single instruction stream is executing and only the data is different on each PE.

10 Applicant's 2nd generation debugger builds on previous experience and provides a variety of ways to visualize the state of PE memory and registers. As well as traditional source-level symbolic views of data, and low-level dumps of memory and data, the debugger provides user-defined *picture* views which show the contents of PE memory or registers in graphical form. This provides a quickly understood view of the overall state of the system.

15 The debugger works identically with all of the simulators and with target hardware. It supports all the features expected from a modern development tool: graphical user interface and command line operation, breakpoints, watchpoints, single-stepping, etc.

Claims

1. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set.
5
2. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted
10 to operate on said scalar data.
3. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein substantially all said instructions are operable on said parallel and/or said
15 scalar data.
4. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of
20 processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.
5. A data processor as claimed in claim 1, wherein said scalar part and said
25 SIMD part are adapted to operate asynchronously.
6. A data processor as claimed in any of claims 1, 3 or 5, wherein the execution unit comprises a SIMD part comprising an array of processing elements (PEs) adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data.

8. A data processor as claimed in claim 6, wherein said SIMD part and said scalar part have a common memory area for instructions and scalar data.
9. A data processor as claimed in claim 6, wherein each PE contains an ALU and one or more multiply-accumulate function units.
10. A data processor as claimed in claim 6, wherein each PE contains an ALU and one or more floating point function units.
11. A data processor as claimed in claim 6, wherein each PE has multiple enable bits to support nested conditional code.
12. A data processor as claimed in claim 6, wherein said PEs and said scalar part each have a similar set of status/result flags which can be used to control branching, on the scalar part of the execution unit, and conditional execution, on the SIMD part of the execution unit, in a similar manner.
13. A data processor as claimed in claim 6, further comprising a communication path between PEs in the SIMD part.
14. A data processor as claimed in claim 6, wherein means are provided for data to be transferred to and from the scalar part of the execution unit and the SIMD part of the execution unit.
15. A data processor as claimed in any of the preceding claims, wherein the control unit is adapted to fetch and execute multiple instruction threads simultaneously.
16. A data processor as claimed in any of the preceding claims, further comprising a semaphore unit adapted to use semaphores to synchronize between instruction threads.
17. A data processor as claimed in claim 16, wherein said semaphores are adapted to synchronize between I/O operations and instruction threads.

18. A data processor as claimed in claim 16, wherein said control unit is adapted to schedule and execute threads based on their priority and the state of any semaphores they are waiting on.

5

19. A data processor as claimed in claim 16, wherein said control unit includes a set of said semaphores adapted to be used to synchronize threads.

20. A data processor as claimed in either of claims 2 or 3, wherein each processing element of the SIMD array comprises one or more function units, a register file, said local memory, and I/O means.

21. A data processor as claimed in any of claims 2, 4, 6 or 7, wherein both said scalar part and said SIMD part have a complete range of addressing modes that allow them to support:

scalar variables pointing to data in scalar memory;
scalar variables pointing to data in memory in the SIMD part of the execution unit;
SIMD variables pointing to memory in the SIMD part of the execution unit; and
SIMD variables pointing to data in scalar memory.

22. A data processor as claimed in any of claims 1, 2 or 5-21, wherein substantially all said instructions are operable on said parallel data or on said scalar data or on a combination of both said parallel data and said scalar data.

25

23. A data processor as claimed in claim 22, further comprising a compiler adapted to produce code from a single source program to operate on both said scalar data and said parallel data.

such that they occur concurrently with computation and data is always available when required.

26. A data processor as claimed in claim 20, wherein said scalar part, said SIMD
5 part and said I/O means are adapted to be operated in parallel by the same instruction stream.

27. A data processor as claimed in claim 26, wherein said I/O means comprises
10 programmed I/O means adapted to enable each PE to transfer data to and from external scalar memory and wherein each PE provides the address it wants to read data from or write data to.

28. A data processor as claimed in claim 27, wherein memory accesses are
15 consolidated to minimize the number of memory transfers when multiple PEs are accessing the same memory area.

29. A data processor as claimed in claim 26, wherein each PE is adapted to
transfer data to and from external scalar memory and wherein the address for each PE
is generated automatically based on the PE number and the amount of data being
20 transferred.

30. A data processor as claimed in claim 26, wherein said I/O means comprises
streaming I/O means which distributes an incoming data stream to all the PEs in the
SIMD part and collects data from all the PEs to generate an output stream
25 independently of program execution on the SIMD part of the execution unit.

31. A data processor as claimed in claim 30, wherein the streaming I/O means
only uses enabled PEs to take part in the streaming I/O operation.

32. A data processor as claimed in claim 30, wherein the size of data distributed
30 to, and collected from each PE is different for each PE.

Abstract

Unified SIMD Processor

5 A data processor comprises a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set. The execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data. Substantially all the instructions are operable on the parallel and/or the
10 scalar data. Preferably, both the scalar part and the SIMD part have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the execution unit; SIMD variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory. The scalar part, the
15 SIMD part, and I/O means are adapted to be operated in parallel by the same instruction stream.

[Figure 2]

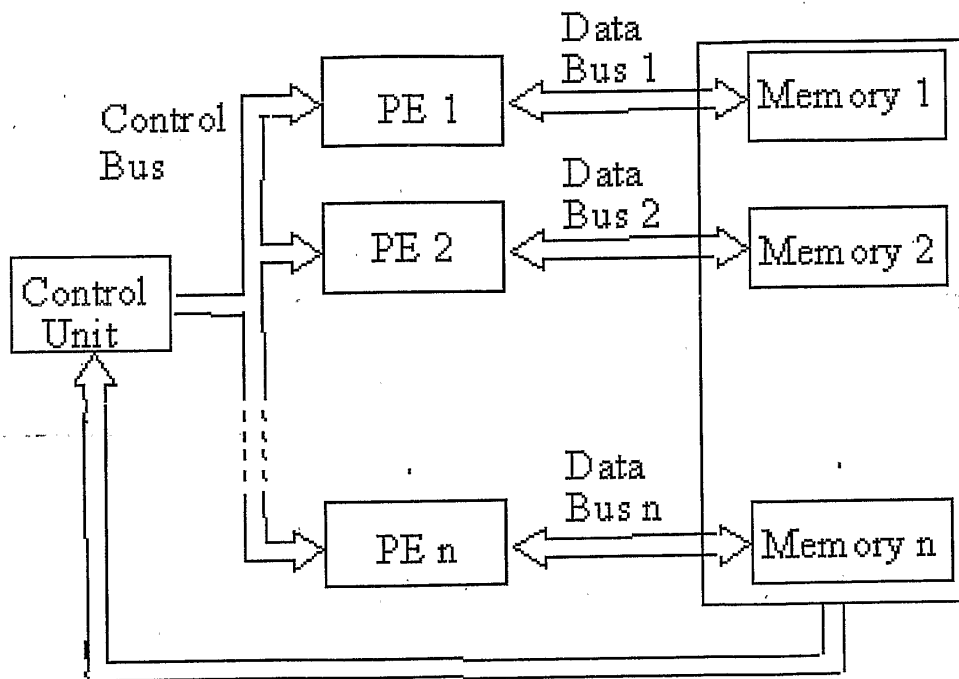


Fig 1



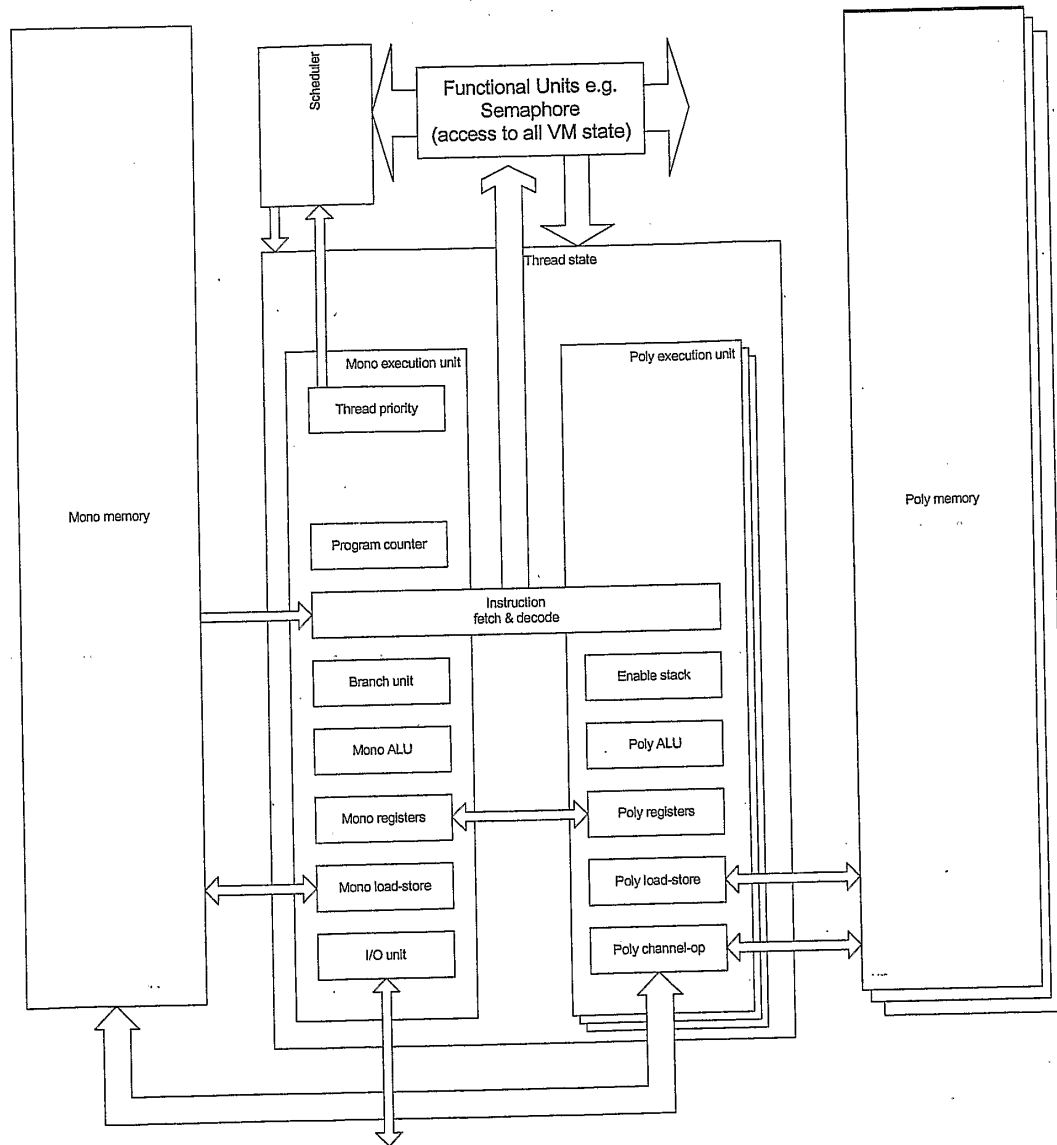


Fig 2

|



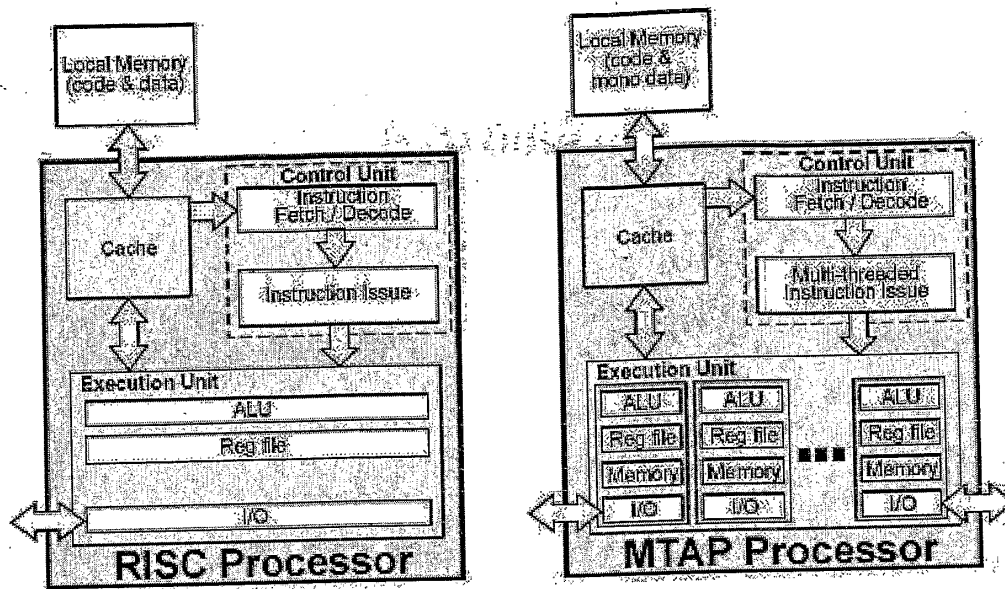


Fig 3

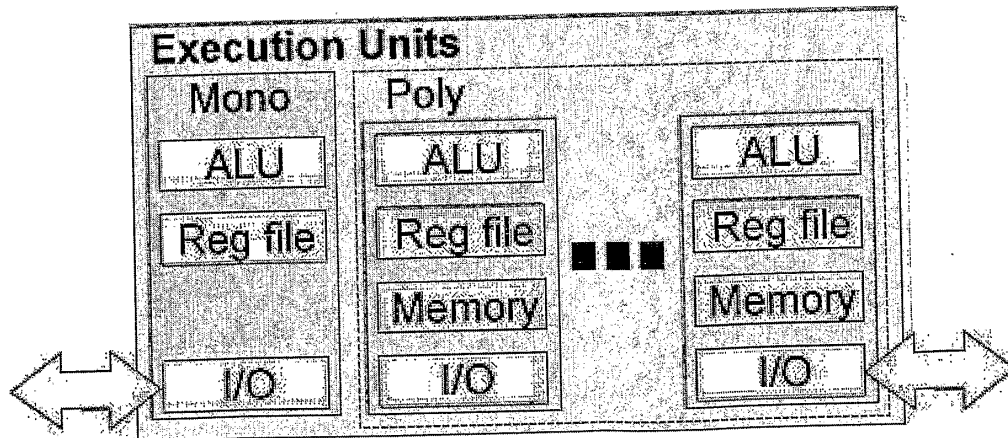


Fig 4

|



1
1
1
1
1

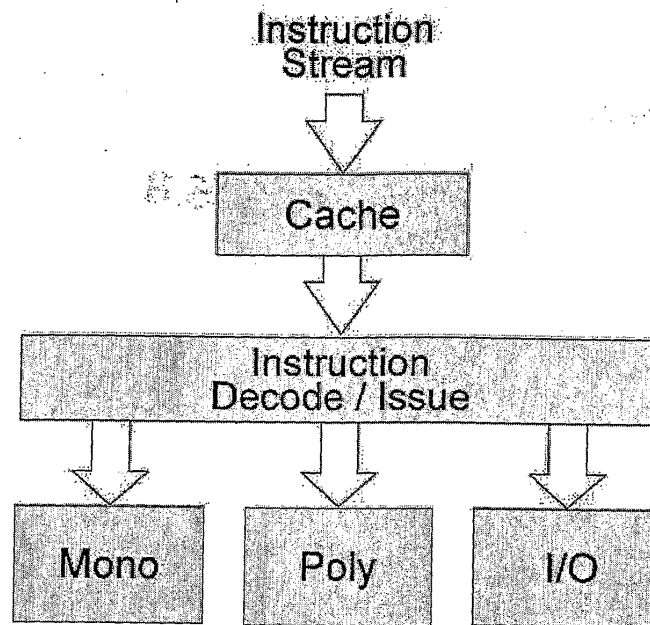


Fig 5

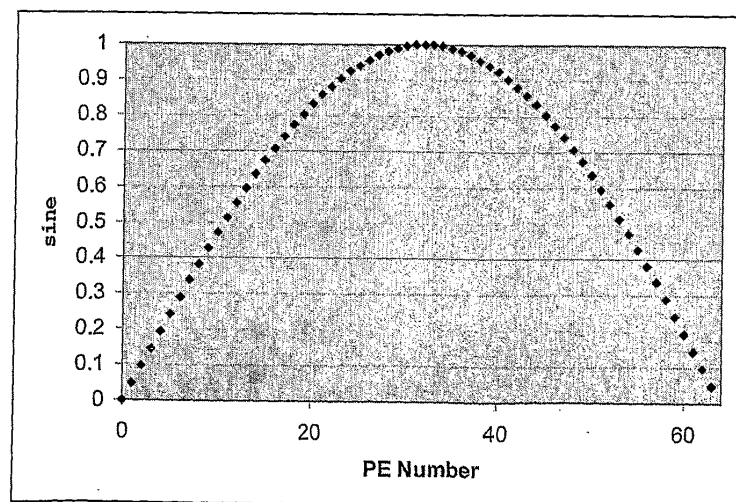


Fig 6

1

2



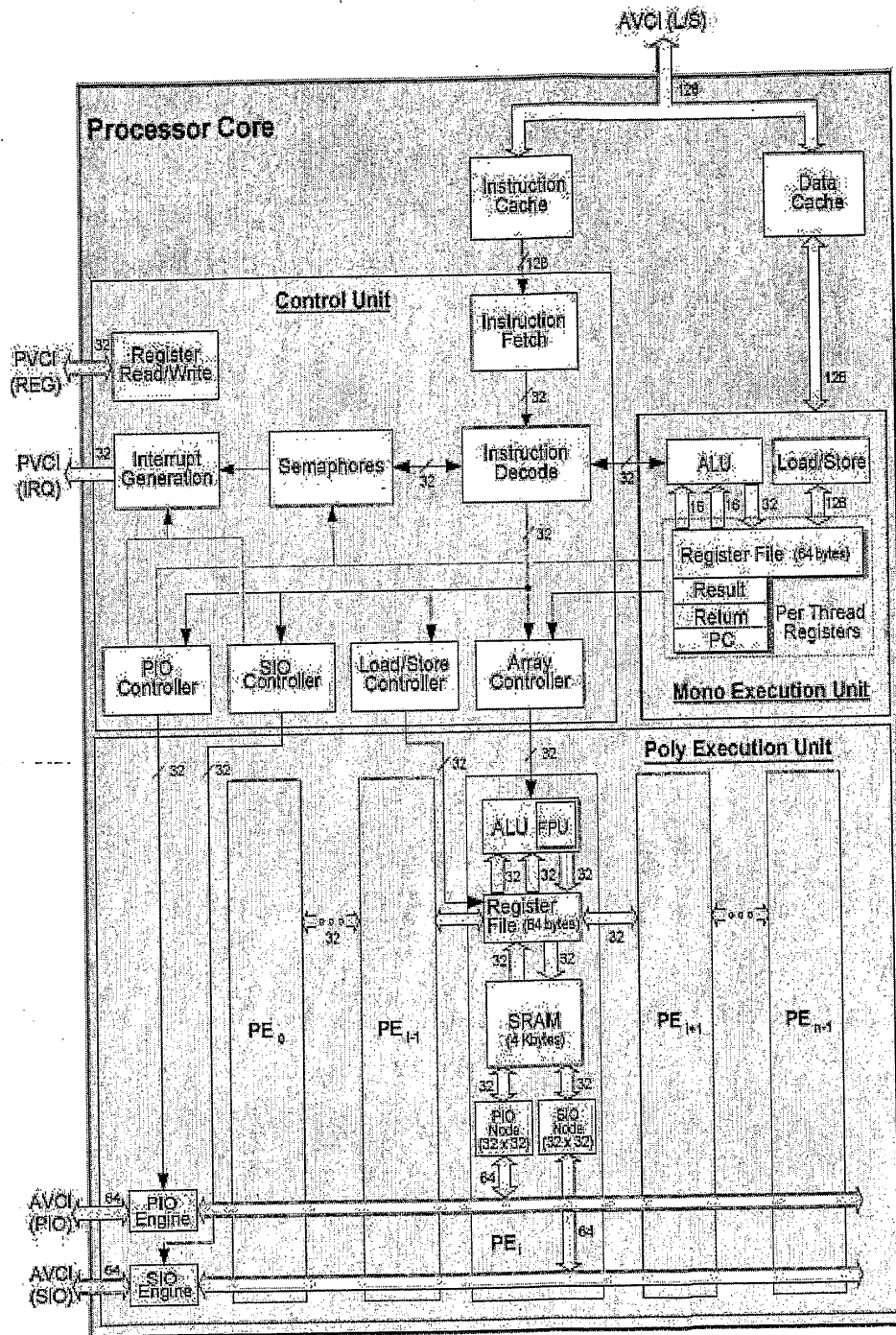


Fig 7

1

2

3



4

5

6

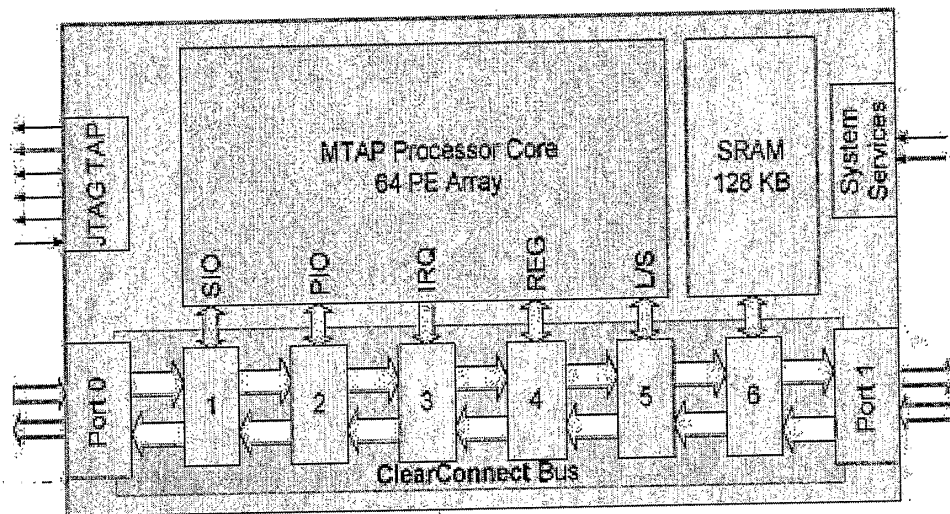


Fig 8

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property
Organization
International Bureau



(43) International Publication Date
28 April 2005 (28.04.2005)

PCT

(10) International Publication Number
WO 2005/037326 A2

(51) International Patent Classification⁷: **A61K 51/04**,
A61P 35/00, A61K 31/16

Bristol BS16 1WR (GB). **DAY, Timothy** [GB/GB]; 33
Rosebery Avenue, St. Werburghs, Bristol BS2 9TW (GB).

(21) International Application Number:
PCT/GB2004/004377

(74) Agent: **VIGARS, Christopher, Ian**; Haseltine Lake, Red-
cliff Quay, 120 Redcliff Street, Bristol BS1 6HU (GB).

(22) International Filing Date: 13 October 2004 (13.10.2004)

(81) Designated States (unless otherwise indicated, for every
kind of national protection available): AE, AG, AL, AM,
AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN,
CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI,
GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE,
KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD,
MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG,
PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM,
TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM,
ZW.

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0323950.6 13 October 2003 (13.10.2003) GB
0409815.8 30 April 2004 (30.04.2004) GB

(84) Designated States (unless otherwise indicated, for every
kind of regional protection available): ARIPO (BW, GH,
GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM,
ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM),
European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI,
FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI,
SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ,
GW, ML, MR, NE, SN, TD, TG).

(71) Applicant (for all designated States except US): **CLEAR-
SPEED TECHNOLOGY PLC** [GB/GB]; 3110 Great
Western Court, Hunts Ground Road, Stoke Gifford, Bristol
BS34 8HP (GB).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **STUTTARD, David**
[GB/GB]; 28 St Georges Avenue, St George, Bristol
BS5 8DD (GB). **WILLIAMS, David** [GB/GB]; 20
Cousins Way, Emersons Green, Bristol BS16 7DG (GB).
PACKER, James [GB/GB]; 43 Whitesfield Road, Nailsea,
Bristol BS48 2DY (GB). **DAVIDSON, Colin** [GB/GB]; 46
Magdelene Street, Glastonbury, Somerset BA6 9EJ (GB).
HICKEY, Neil [GB/GB]; 34 Casson Drive, Stoke Park,

Published:

— without international search report and to be republished
upon receipt of that report

For two-letter codes and other abbreviations, refer to the "Guid-
ance Notes on Codes and Abbreviations" appearing at the begin-
ning of each regular issue of the PCT Gazette.

(54) Title: UNIFIED SIMD PROCESSOR

(57) Abstract: A data processor comprises a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set. The execution unit comprises a SMO part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data. Substantially all the instructions are operable on the parallel and/or the scalar data. Preferably, both the scalar part and the SIMD part have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the execution unit; SHVM variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory. The scalar part, the SIMD part, and I/O means are adapted to be operated in parallel by the same instruction stream.

WO 2005/037326 A2

Unified SIMD Processor

Background to the Invention

The invention consists of a significant improvement to a well known type of parallel processor. Single-Instruction Multiple-Data (SIMD) architectures are widely used in a number of computing applications. They have the advantage of high performance, low power consumption and easy scalability. A SIMD processor differs from the more commonly used multi-processor system (where a number of processors, each independently executing their own program code, are used together to solve a problem) in that it executes a single stream of instructions but each instruction operates on multiple data items in parallel.

There are a range of SIMD processor architectures. These range from an *intelligent memory* type of approach where each *column* of a memory array has a simple 1-bit Arithmetic Logic Unit (ALU) through to more complex architectures where a, usually smaller, array of more complex processing elements are fed with a single instruction stream. They have, however, been difficult to program in the past.

In all these cases the array of SIMD processing elements are fed with instructions and data by a dedicated controller. Often the controller is the largest and most complex part of the design because of the need to match the data processing rate achieved by the SIMD array. The array controller is not itself 'intelligent' and is, in turn, controlled by a general purpose processor which also runs the bulk of the application software. This *host* processor will execute most of the application code; it will send data to be processed to the SIMD array with an indication of the function to be performed, typically a pointer to the start of a sequence of SIMD array instructions or microinstructions. An exemplary prior art SIMD processor is outlined in Figure 1.

This introduces two programming problems. Firstly, the array controller can only be programmed at a very low level: a series of microcode, or possibly assembler, instructions to be executed by the array. This is because the array controller is not designed to work with a compiler. Secondly, this programming is completely separated from the main application code running on the host processor. This may be 'wrapped up' by providing C functions or C++ classes which makes them appear more integrated and reduces the complexity of passing control between the host processor and the SIMD array. Also, in the simplest form, the host processor and the SIMD array will have completely separate memory spaces for code and data. Any of

the standard shared memory techniques can be used to address this with all the resulting problems: complex arbitration, cache coherency issues, contention for memory bandwidth, inter-processor synchronization, greater cost, etc.

A preferable solution is to provide a fully integrated architecture where the same processor can transparently process both *scalar* and SIMD operations in a fully integrated fashion. This means that a single program can be written, in a standard high-level language such as C, which includes all parts of the application – both ‘normal’ (scalar) data and parallel data processing.

Prior Art

The basic SIMD processing model has been in use for many years. Some of the earliest examples are the ILLIAC IV developed in 1972, which had 64 processors with floating point and memory, and Goodyear’s STARAN (1975) which was a 1-bit architecture. Many variants have been developed since then.

There is also a recent trend to put small-scale SIMD extensions in standard processor architectures. This is exemplified by the Intel MMX and SSE extensions to the x86 architecture. This allows, for example, the values in two 32-bit registers to be added as if each register contained four independent 8-bit values. This is not considered relevant to the current discussion for a number of reasons:

- the SIMD processing is done on a very small scale;
- only a limited range of SIMD functionality is provided;
- there are distinct instructions for operating on SIMD data;
- the same hardware is used for SIMD and for scalar operations (although separate registers may be provided for the SIMD operands)

The idea of providing indexed addressing modes in a SIMD processor is disclosed in our copending patent application GB9908225.7. This invention extends that idea by using a more flexible implementation of that idea in combination with a more powerful array controller to provide a unified processor architecture.

Summary of the Invention

Applicant’s approach, after many years of experience with developing systems based on SIMD coprocessors, was to create a new architecture where instructions are decoded and dispatched to the appropriate part of the execution unit: either the scalar (mono) execution unit or the SIMD (poly) execution unit. Instruction execution on these are decoupled so that, for example, a mono instruction followed by a poly instruction will execute concurrently. The support for mono and poly data is, as far as

possible, identical – this makes it practical to write a compiler to target the processor. This greatly simplifies the task of programming the processor.

This means a single program runs on the processor which does both mono and poly operations as required – typically it will do both at the same time. A single C program
5 can be written which manages program control flow, operates on mono data, and operates on poly data.

To support this, and further simplify programming, this unity of the architecture is reflected in the instruction set and programming model. The processing elements in the SIMD array and the mono execution unit have a common instruction set. For
10 instance, an **add** instruction can be used with operands which are either poly registers or mono registers (or immediate values); the appropriate execution unit will execute the instruction. Mono and poly operands can also be mixed to allow, for example, a single mono value to be added to a poly register which holds a different value on every processing element.

15 It is appropriate to note that this does not require the mono execution unit to have exactly the same micro-architecture as the poly execution unit. For example, the poly processing elements could have an 8-bit ALU, while the mono execution unit has a 32-bit ALU.

A block schematic diagram of a processor in accordance with the invention is shown
20 in Figure 2.

The most significant features contributing to the novelty and inventiveness of this invention

include the following concepts:

- The processor provides a unified programming model: a single program runs on
25 a processor providing both scalar and SIMD operations.
- This is supported with a unified instruction set: e.g. an **add** instruction can work with either mono or poly operands (or a mixture of the two)
- Poly operations can freely use mono registers in expressions, and more importantly for addressing memory.
- 30 • Having a full range of addressing modes and completely separate memory blocks in each PE make it simple to support poly data structures and poly pointers on the SIMD array (previous SIMD architectures either provide a single address to all PEs or use sequences of instructions to construct an address in memory).

- Mono and poly operations are of equal standing: wherever possible, all instructions can be used with either mono or poly operands (there are some exceptions because a small set of operations, mainly for control, only make sense for one or the other).
 - 5 • The mono and poly execution units are loosely coupled so that their operation can be overlapped. Compilers are extremely good at scheduling instructions to take advantage of this flexibility - maximizing the efficiency of the system.
 - There are some compiler optimizations specific to this architecture: for example it may be better, in some instances, to move an operation that would normally
10 be done on the mono execution unit on to the poly execution unit - either because it can then be overlapped with other operations on the mono execution unit or because the result is needed on the poly execution unit anyway. Again, the compiler can detect such cases and make the appropriate optimizations.
 - A common code and data space is available to both mono and poly operations
15 making it easy to generate code for, and share data between, the two execution units.
 - The overall architecture is suitable for targeting by a high level language compiler - again for both mono and poly operations.
- 20 To this end, the invention provides, in a first aspect a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set.
- According to a second aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel
25 and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data.
- According to a third aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel
30 and scalar data under the same instruction set, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.
- According to a fourth aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit

SUBSTITUTE SHEET (RULE 26)

comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.

- 5 The single control unit and the integral execution unit may be adapted to operate asynchronously on both the parallel and scalar data under the same instruction set.

The execution unit preferably comprises a SIMD part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data.

- 10 Preferably, the SIMD part and the scalar part differ only in that each processing element of the array contains local memory. The SIMD part and the scalar part preferably have a common memory area for instructions and scalar data. Each PE may contain an ALU and one or more multiply-accumulate function units or an ALU and one or more floating point function units. Preferably, each PE has multiple
15 enable bits to support nested conditional code.

The SIMD and the scalar execution unit conveniently have a similar set of status/result flags which can be used to control branching on the scalar part of the execution unit and conditional execution on the SIMD part of the execution unit in a similar manner.

- 20 There is preferably a communication path between PEs in the SIMD part and means are preferably provided for data to be transferred to and from the scalar part of the execution unit and the SIMD part of the execution unit.

Advantageously the control unit is adapted to fetch and execute multiple instruction threads simultaneously. A semaphore unit may use semaphores to synchronize
25 between instruction threads. The semaphores may be used to synchronize between I/O operations and instruction threads. The control unit may be adapted to schedule and execute threads based on their priority and the state of any semaphores they are waiting on and may include a set of the semaphores for use in synchronizing threads.

- Each processing element of the SIMD array may comprise one or more function units,
30 a register file, said local memory, and I/O means.

Both the scalar part and the SIMD part preferably have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the

execution unit; SIMD variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory.

Substantially all the instructions are preferably operable on either the parallel data or the scalar data or a combination of both the scalar data and the parallel data.

- 5 The processor may further comprise a compiler adapted to produce code from a single source program to operate on both the scalar and the parallel data. The compiler is preferably adapted to detect parallel data automatically.

The scalar part, the SIMD part, and the I/O means are preferably adapted to be operated in parallel by the same instruction stream.

- 10 A programmer or the compiler is adapted to use the multi-threaded execution to schedule I/O operations such that they occur concurrently with computation and data is always available when required.

- The I/O means may comprise programmed I/O means adapted to enable each PE to transfer data to and from external scalar memory and wherein each PE provides the address it wants to read data from or write data to. The memory accesses may be consolidated to minimize the number of memory transfers when multiple PEs are accessing the same memory area.

- 15 Each PE is preferably adapted to transfer data to and from external scalar memory, in which case the address for each PE is generated automatically based on the PE number and the amount of data being transferred.

- The I/O means may comprise streaming I/O means which distributes an incoming data stream to all the PEs in the SIMD part and collects data from all the PEs to generate an output stream independently of program execution on the SIMD part of the execution unit. The streaming I/O means may only use enabled PEs to take part in the streaming I/O operation. The size of data distributed to, and collected from each PE may be different for each PE.

Brief Description of the Drawings

The invention will be described with reference to the following drawings, in which:

- Figure 1 shows a prior art SIMD processor;
- 30 Figure 2 shows the architecture in accordance with the present invention;
- Figure 3 is a comparison between RISC and MTAP architecture;
- Figure 4 is a schematic diagram of an MTAP execution unit;
- Figure 5 indicates schematically the execution of an instruction;
- Figure 6 is a graph indicating the power efficiency of the MTAP architecture;

Figure 7 is a block schematic diagram depicting an exemplary MTAP processor; and

Figure 8 is a block diagram of a specific implementation of the MTAP architecture as applied to an evaluation chip.

5 Detailed Description of the Illustrated Embodiments

The design of this processor architecture was driven by the desire to make it practical and efficient to program in a high level language. In order to make this possible a number of features are required. First, the overall programming model needs to be simple and 'regular' – i.e. the same operations and addressing modes can be applied to
10 all data types.

The execution unit for mono data is based on well known principles used in many RISC processors. This makes it simple for a compiler to target and easy for a user to understand.

The functionality supported by each PE is then made as similar as possible to the mono execution unit: each PE has an ALU, a register file and memory, and supports a
15 range of addressing modes for transferring data between memory and registers.

It is not necessary for the PEs to be identical to the mono execution unit in every detail: e.g. the number of registers and the width and complexity of the ALU are likely to be different in practice. The poly execution unit gets its performance largely
20 from the number of PEs brought to bear on the problem rather than the processing power of the individual PEs. To balance this, the mono execution unit is likely to require a wider and more complex ALU. This is reasonable because this is only instantiated once while the PEs are replicated many times and need to be efficiently implemented.

The number of clock cycles required to execute an instruction on the mono and poly execution units is highly variable. Both will execute some instructions in a single cycle. But some instructions could take several cycles on the mono execution unit or on the poly execution unit. It is therefore essential that the two execution units are only loosely coupled rather than operating in lock-step; this allows them to overlap
30 their execution of instructions. The hardware will synchronize their operations when necessary: e.g. when they need to access a shared resource, or one execution unit requires data from the other.

Transfers between the registers and memory, and other I/O transfers, can take place concurrently with ALU operations. To simplify the task for the compiler, these

interactions are all interlocked using register scoreboarding (see our copending patent application GB 9908203.4).

An important aspect of the PE architecture to support compiled code is the existence of a full set of addressing modes: direct, indirect and indexed. Indirect and indexed
5 addressing by the PEs is described in the previously mentioned copending patent application GB 9908225.7.

Reference may also be had to our copending patent application GB 0321186.9 for details of the "ClearConnect" bus referred to in the context of the present invention.

In a high level language compiler for this architecture, it is only necessary to
10 introduce a simple modification; the keyword 'poly' used in a declaration identifies data that is distributed across the SIMD array. Similar methods have been used before, for example the 'shape' keyword used in C*, a language used for programming the "Connection Machine". However, it is only with the present architecture that poly variables become 'first class' objects – i.e. they can be used everywhere that 'normal'
15 mono data can be: in expressions, as function arguments, in conditional statements, etc. Pointers to both mono and poly data are allowed, as are pointers of both mono and poly types. Furthermore, poly variables can be mixed quite freely with mono data in all these contexts. This provides a very simple and regular programming model for the user.

20 Note that it would also be possible for a compiler to detect the parallelism in the program or data automatically: This is frequently done with 'vectorizing' Fortran compilers used with supercomputers, for example. The same techniques could be applied to this architecture. The architecture will now be described in more detail.

The Multi-Threaded Array Processing (MTAP) architecture has been developed to
25 address a number of problems in high performance, high data rate processing.

A number of applications are characterized by very high data rates, flexible processing requirements and hard real-time constraints. We use the term *data flow processing* to describe this class of problem.

The MTAP processor delivers on the three primary requirements for data flow
30 applications:

1. It directly addresses the data bandwidth, and has a clear scalability path for future requirements.

2. It provides the raw horsepower for the processing functions required, on the maximum data rate that the system will encounter. That processing power can scale with increasing function demands.
3. It stores the data close to the processing core to maximize bandwidth and minimize latency.

MTAP Overview

The MTAP architecture defines a family of embedded processors with parallel data processing capability. Figure 3 compares a standard processor and the MTAP architecture. As can be seen, the MTAP processor has a standard, RISC-like, control unit with instruction fetch, caches and I/O mechanisms. This is coupled to a highly parallel execution unit which provides the performance and scalability of the MTAP architecture.

- The processor is designed in a highly modular fashion which allows many details of a specific implementation to be easily customized for the target application.
- To simplify the integration of the processor into a variety of systems, the processor can be also configured to be big or little-endian.

Control unit

The control unit fetches, decodes and dispatches instructions to the execution units. The processor executes a fairly standard, three operand instruction set. The manner in which the same instruction stream is decoded and issued in parallel to the mono part, the poly part and the I/O part is schematically indicated in Figure 5.

- The control unit also provides hardware support for multi-threaded execution, allowing fast swapping between multiple threads. The threads are prioritized and are intended primarily to support efficient overlap of I/O and compute. This can be used to hide the latency of external data accesses.

The MTAP processor can also include instruction and data caches to minimize the latency of memory accesses. The size and type of these is a configurable option. Alternatively, the caches may be replaced with SRAM to provide a complete embedded solution.

- The control unit also includes a control port which is used for initializing and debugging the processor; it includes support for breakpoints, single stepping and the examination of internal state.

Execution unit

The execution unit consists of a number of *processing elements* (PEs). This allows it to process data elements in parallel. Each PE consists of an ALU, registers, memory and I/O.

- 5 The number of PEs in a processor core is a configurable parameter which allows performance to be scaled to meet the needs of the application (see Figure 6). The execution unit can be thought of as two, largely independent, parts (see Figure 4).

One PE forms the *mono* execution unit; this is dedicated to processing mono (i.e. scalar or non-parallel) data. The mono execution unit also handles program flow
10 control such as branching and thread switching. The rest of the PEs form the *poly* execution unit which processes parallel (poly) data.

The poly execution unit may consist of tens, hundreds or even thousands of PEs. This array of PEs operates in a synchronous manner, similar to SIMD, where every PE executes the same instruction on its piece of data. Each PE also has its own
15 independent local memory; this provides fast access to the data being processed. For example, one PE at 400 MHz has a memory bandwidth of 1.6 Gbytes/s. An array with 256 such PEs has an aggregate bandwidth of over 400 Gbytes/s, with single cycle latency. The number of registers and the amount of memory in each PE are configurable options.

20 **Input / output**

The MTAP processor core has two basic I/O mechanisms. The first, *Programmed I/O* (PIO), is the normal mechanism used for accessing memory external to the MTAP core: it supports random accesses to variable sized data by the PEs. The second mechanism is *Streaming I/O* (SIO) which allows chunks of contiguous data to be
25 streamed directly into the memory within PEs.

Each of these supports a variety of addressing modes, described in more detail later.

Programming model

From a programmer's perspective, the MTAP processor appears as a single processor running a single C program. This is very different from some other parallel processing
30 models where the programmer has to explicitly program multiple independent processors, or can only access the processor via function calls or some other indirect mechanism.

The MTAP processor executes a single instruction stream; each instruction is sent to one of the functional units: this may be the mono or poly execution unit or one of the

I/O controllers. The processor can despatch an instruction on every cycle.

For multi-cycle instructions, the operation of the functional units can be overlapped.

So, for example, an I/O operation can be started with one instruction and on the next cycle the mono execution unit could start a multiply instruction (which requires

5 several cycles to execute). While these operations are proceeding, the poly execution unit can continue to execute instructions.

The main change from programming a standard processor is the concept of operating on parallel data. Data to be processed is assigned to variables which have an instance on every PE and are operated on in parallel: we call this *poly* data. This can be
10 thought of as a data vector which is distributed across the array.

Variables only requiring a single instance (e.g. loop control variables) are known as *mono* variables – they behave exactly like normal variables on a sequential processor.

Applicant provides a compiler which uses a simple extension to standard C to identify data which is to be processed in parallel. The new keyword **poly** is used in a

15 declaration to define data which exists, and is processed, on every PE in the poly execution unit.

Example

To give a feel for the way the MTAP processor is programmed, the following represents a fragment of code which will calculate 64 values of a sine function across

20 the PE array in a single operation.

```
#include <fnext.h>
#include <math.h>
#define PI 3.1415926535897932384
#define NUMBER_OF_PES 64

25 poly float angle, sine;
poly int pe;

/* get PE number: 0...n-1 */
30 pe = get_penum();
/* convert to an angle in range 0 to Pi */
angle = pe * PI / NUMBER_OF_PES;
/* calculate sine of angle on each PE */
sine = sinp(angle);
```

SUBSTITUTE SHEET (RULE 26)

This code uses the library function `get_penum()` to get a unique value in the variable `pe` on each PE. This is scaled to give a range of values for `angle` between 0 and π across the PEs.

Finally, the library function `sinp()` is called to calculate the sine of these values on all PEs simultaneously. The `sinp` function is the poly equivalent of the standard sine function; it takes a poly argument and returns the appropriate value on every PE.

Performance

The performance of an MTAP core with 64 PEs, on a 0.13 μ process and running at 400 MHz is shown in Table 1. The columns show the performance achieved with the basic PE, the PE with an additional Multiply-Accumulate (MAC) unit, and the PE with an FPU extension.

Table 1 - Performance Comparison

	Base Architecture	8-bit MAC	FPU
MIPS	25,600	25,600	25,600
MMACS ($\times 10^6$)	—	25,600	25,600
MFLOPS	—	—	51,200
Memory Bandwidth (Gbytes / s)	100	100	100
Inter-PE Bandwidth (GBytes / s)	50	50	50
Relative Area	1.0	1.1	1.3

However, the core can be scaled well beyond this number of PEs, to 256 or even thousands of PEs. It also scales down efficiently to lower numbers of PEs, or lower clock rates, enabling low cost and low power systems to be built.

Figure 6 shows how performance and power scale with the number of PEs. By choosing the number of PEs and the clock frequency, the designer has a great deal of flexibility to tune performance, bandwidth, cost and power for the needs of a given application.

Architecture Details

The following sections describe the MTAP processor in more detail. A block diagram of the processor is shown in Figure 7. This is a conceptual view, which reflects the programming model. In practice, the control and mono execution units form a tightly-coupled pipeline which provides overall control of the processor.

Interfaces

The processor has a number of external interfaces. The widths of each of these are configurable. All these interfaces currently use the Virtual Component Interface Standard defined by the VSI Alliance.

There are three categories of interface:

Mono data & instructions

This is a single *Advanced* VCI (AVCI) interface which is used for mono loads and stores, and for instruction fetching.

Poly data

This uses one or more AVCI interfaces for PIO and SIO data. The number of physical interfaces corresponds to the number of PIO and SIO channels implemented.

Control

There are two control interfaces which use the *Peripheral* VCI (PVCi) protocol. One is used for initialization and debug. The other allows the MTAP processor to generate interrupts to the host system.

Instruction set

The processor has a fairly standard RISC-like instruction set. Most instructions can operate on mono or poly operands and are executed by the appropriate part of the execution unit. Some instructions are only relevant to either the mono or poly execution unit, for example all program flow control is handled by the mono unit.

The instruction set provides a standard set of functions on both mono and poly execution units:

- Integer and floating point adds, subtracts, multiplies, divides
- Logical operations: and, or, not, xor
- Arithmetic and logical shifts
- Data comparisons: equal, not equal, greater than, less than, etc.
- Data movement between registers
- Loads and stores between registers and memory

To give a feel for the nature of assembly code for the MTAP processor, a few lines of code are shown below. This simple example loads a value from memory into a mono register, gets the PE number into a register on each PE and then adds these two values together producing a different result on every PE.

```

5  ld 0:m4, 0x3000    // load mono reg 0 from mem
   penum 8:p4         // PE number into poly reg
   add 4:p4, 0:m4, 8:m4 // add; result in reg 4

```

Control unit

The control unit fetches instructions from memory, decodes them and despatches them to the appropriate functional unit.

The controller includes a scheduler to provide hardware support for multi-threaded code. This is a vital part of the architecture for achieving the performance potential of the MTAP processor. Because of the highly parallel architecture of the poly execution unit, there can be significant latencies if all PEs need to read or write external data.

When part of an application stalls because it is waiting for data from external memory, the processor can switch to another code thread that is ready to run. This serves to hide the latency of accesses and keep the processor busy.

The number of threads is a configurable option. The threads are prioritized: the processor will run the highest priority thread that is ready to run; a higher priority thread can pre-empt a lower priority thread when it becomes ready to run. Threads are synchronized – with each other and with hardware, such as I/O engines – via semaphores.

In the simplest case of multi-threaded code, a program would have two threads: one for I/O and one for compute. By *pre-fetching* data in the I/O thread, the programmer (or the compiler) can ensure the data is available when it is required by the execution units and the processor can run without stalling.

Execution units

This section describes the common aspects of the poly and mono execution units.

ALU operations

- Instructions for arithmetic and logical operations are provided in several versions:
- Various sizes: 1, 2, 3 and 4 bytes, and combinations of sizes
 - Various data types: signed and unsigned integers, and floating-point
 - Mixes of signed/unsigned operations

Status register

Associated with the ALU is a status register; this contains five status bits that provide information about the result of the last ALU operation. When set, these bits indicate:

- Most significant bit set
- 5 • Carry generated
- Overflow generated
- Negative result
- Zero result

Registers

- 10 To support operations on data of different widths, the registers in the PEs can be accessed very flexibly. The register files are best thought of as an array of bytes which can be addressed as registers of 1 to 8 bytes wide.

- The mono and poly registers are addressed in a consistent way using byte addresses and widths specified in bytes. The mono register file is 16 bits wide and so all
15 addresses and widths must be a multiple of 2. There are no alignment restrictions on poly register accesses.

Addressing modes

- Load and store instructions are used to transfer data between memory and registers. In the case of the mono execution unit, these transfer data to and from memory external
20 to the MTAP processor.

Poly loads and stores transfer data between the PE register file and PE memory. Data is transferred between the PEs and external memory using the I/O functions described later.

- There are three addressing modes for loads and stores which can be used for both
25 mono and poly data. These are:

Direct

The address to be read/written is specified as an immediate value.

Indirect

The address is specified in a register.

- 30 **Indexed**

The address is calculated from adding an offset to a base address in a register. The offset must be an immediate value.

Conditional code

The main difference between code for the mono and the poly execution units is the handling of conditional execution.

The mono unit uses conditional jumps to branch around code, typically based on the result of the previous instructions. This means that mono conditions affect both mono and poly operations. This is just like a standard RISC architecture.

The poly unit uses a set of enable bits (described in more detail below) to control whether each PE will have its state changed instructions it executes. This provides per-PE predicated operation. The enable state can be changed based on the result of a previous operation.

The following sections describe the architectural features of the two execution units in more detail.

Mono execution unit

The mono execution unit is a 16-bit processing element consisting of:

- A 16-bit ALU with optional extensions
- A register file of configurable size
- Status and control registers

The mono ALU extensions include a multiplier, a barrel shifter and a normalizer, which is important for accelerating software implementations of floating point operations.

As well as handling mono data, the mono unit is responsible for program flow control (branching), thread switching and other control functions. The mono execution unit also has overall control of I/O operations of poly data. Results from these operations are returned to a register in the mono unit.

Conditional execution

The mono execution unit handles conditional execution in the same way as a traditional processor. A set of conditional and unconditional jump instructions use the result of previous operations to jump over conditional code, back to the start of loops, etc.

Multi-threaded execution

The MTAP processor supports several hardware threads. There is a hardware scheduler in the control unit and the mono execution unit maintains multiple banks of critical registers for fast context switching.

The threads are prioritized (0 being highest priority). Control of execution between the threads is performed using semaphores under programmer control. Higher priority threads will only yield to lower priority threads when they are stalled on yielding instructions (such as semaphore wait operations). Lower priority threads can be pre-empted at any time by higher priority ones.

- 5 Semaphores are special registers that can be incremented or decremented with atomic (non-interruptible) operations called *signal* and *wait*. A *signal* instruction will increment a semaphore. A *wait* will decrement a semaphore unless the semaphore is 0, in which case it will stall until the semaphore is signalled by another thread.
- 10 Semaphores can also be accessed by hardware units (such as the I/O controllers) to synchronize these with software.

Poly execution unit

The poly execution unit is an array of Processing Elements (PEs). Each PE in the poly execution unit consists of:

- 15
- An 8-bit ALU with optional extensions such as a multiply-accumulate (MAC) unit
 - A register file of configurable size
 - Status and enable registers
 - A block of memory of configurable size
 - An inter-PE communication path
- 20
- One or more I/O channels

Load and store instructions move data between a PE's register file and memory, while the ALU operates on data in the register file. Data is transferred in to, and out of, the PE's memory using I/O instructions.

ALU

- 25 The poly ALU is used for performing arithmetic and logical operations on values held in the PE register file.

While the ALU is only 8 bits wide, instructions exist for multi-byte arithmetic which is handled by iteration. ALU extensions exist to accelerate functions for floating point, DSP, etc. For example, an optional integer multiply-accumulate unit (MAC)

30 can be included in the ALU. This can deliver an 8 x 8 bit MAC result every cycle, or a 16 x 16 bit MAC every four cycles. The accumulated result can be up to 64 bits wide. The standard ALU includes basic hardware support to accelerate floating point operations.

Conditional behaviour

The SIMD nature of the PE array prohibits each PE having its own branch unit (branching being handled by the mono execution unit). Instead, each PE can control whether its state should be updated by the current instruction by *enabling* or *disabling* itself; this is rather like the predicated instructions in some RISC CPUs.

Enable state

A PE's enable state is determined by a number of bits in the enable register. If all these bits are set to one, then a PE is *enabled* and executes instructions normally. If one or more of the enable bits is zero, then the PE is *disabled* and most instructions it receives will be ignored (instructions on the enable state itself, for example, are not be disabled).

The enable register is treated as a stack, and new bits can be pushed onto the top of the stack allowing nested predicated execution. The result of a test, either a 1 or a 0, is pushed onto the enable stack. This bit can later be popped from the top of the stack to remove the effect of that condition. This makes handling nested conditions and loops very efficient. Note that, although the enable stack is of fixed size, the compiler handles saving and restoring the state automatically, so there are no limitations on compiled code. When programming at the assembler level, it is the programmer's responsibility to manage the stack.

Instructions

Conditional execution on the poly execution unit is supported by a set of poly conditional instructions: **if**, **else**, **endif**, etc. These manage the enable bits to allow different PEs to execute each branch of an **if...else** construct in C, for example. These also support nested conditions by pushing and popping the condition value on the enable stack.

As a simple example, consider the following code fragment:

```
// disable PEs where reg 32 is non-zero
if.eq 32:p1, 0 // push result onto stack
// increment reg 8 on enabled PEs
30 add    8:p4, 8:p4, 1
// return all PEs to original enable state
endif           // pop enable stack
```

Here, the initial **if** instruction compares the two operands on each PE. If they are equal it pushes 1 onto the top of the enable stack – this leaves those PEs enabled if they

were previously enabled and disabled if they were already disabled. If the two operands are not equal, a 0 is pushed onto the stack – this disables the corresponding PEs.

The following **add** instruction is sent to all PEs, but only acted on by those that are still enabled. Finally, the **endif** instruction pops the enable stack, returning all PEs to their original enable state.

Forced loads and stores

Poly loads and stores are normally predicated by the enable state of the PE. However, because there are instances where it is necessary to load and store data regardless of the current enable state, the instruction set includes *forced* loads and stores. These will change the state of the PE even if it is disabled.

I/O mechanisms

There are two I/O mechanisms provided for transferring data between PE memory and devices outside the MTAP core. *Programmed I/O* (PIO) extends the load/ store model: it is used for transfers of small amounts of data between PE memory and external memory. *Streaming I/O* (SIO) is used to efficiently transfer contiguous chunks of data in and out of PE memory.

The number of PIO and SIO channels in a processor is a configurable parameter. A processor will always have at least one PIO channel as this is required for normal program execution.

Multiple I/O channels can run simultaneously.

I/O architecture

The I/O systems consist of three parts: Controller, Engine and Node.

Controller

The PIO and SIO controllers decode I/O instructions and coordinate with the rest of the control unit and the mono processor. The controllers synchronize with software threads via semaphores.

Engine

The I/O engines are basically DMA engines which manage the actual data transfer.

There is a Controller and Engine for each I/O channel. A single Controller can manage several I/O Engines.

Node

There is an I/O Node in each PE. The I/O Engine activates each Node in turn allowing to serialize the data transfers. The Nodes provide buffering of data to minimize the impact of I/O on the performance of the PEs.

5 **Programmed I/O (PIO)**

PIO is closely coupled to program execution and is the normal way for the processor to transfer data to and from the outside world (e.g. external memory, hardware accelerators or a host processor).

The PIO mechanism provides a number of addressing modes:

10 **Direct addressed**

Each PE provides an external memory address for its data. This provides random access to data.

Strided

15 The external memory address is incremented for each PE. In each case, the size of data transferred to each PE is the same.

When multiple PEs are accessing memory then the transfers can be *consolidated* so as to perform the minimum number of external accesses. So, for example, if half the processors are reading one location and the other half reading another, then only two memory reads would be performed. In fact, consolidation can be better than that:
20 because the bus transfers are *packetized*, even transfers from nearby addresses can be effectively consolidated.

Streaming I/O (SIO)

SIO is used for streaming high-bandwidth data directly to and from PE memory. This is less flexible but very efficient for transferring blocks of data in and out of the
25 system. This is typically used to stream data to and from memory mapped I/O devices. Each PE can transfer a different size block of data.

Streaming I/O can be made very efficient in a number of ways. For example, multi-threaded code allows data I/O and compute to be fully overlapped. Also, multiple data buffers can be allocated in PE memory so that different chunks of data can be input,
30 processed and streamed out simultaneously.

Swizzle

Finally, the PEs are able to communicate with one another via what is known as the *swizzle* path that connects the register file of each PE with the register files of its left and right neighbours. On each cycle, PE_n can perform a register-to-register transfer of

16 bits to either its left or right neighbour, PE_{n-1} or PE_{n+1} , while simultaneously receiving data from the other neighbour.

Swazzle instructions use multiples of 2 bytes in their arguments, so the source and destination registers must be 2-byte aligned. Instructions are provided to shift data left

5 or right through the array, and to swap data between adjacent PEs.

The enable state of a PE affects its participation in a swazzle operation in the following way: if a PE is enabled, then its registers may be updated by a neighbouring PE, regardless of the neighbouring PE's enable state. Conversely, if a PE is disabled, its register file will not be altered by a neighbour under any circumstance. A disabled

10 PE will still provide data to an enabled neighbour.

The data written into the registers of the PEs at the ends of the swazzle path can be set by the mono execution unit.

Host interface

The interfaces to the host system are used for 3 basic purposes: initialization and
15 booting, access to host services and debugging.

Initialization

There are a number of stages of initialization required to start code running on the MTAP processor. These are normally handled transparently by the development tools, but an overview is provided here as background information.

20 First, the application code (including bootstrap) is loaded into memory.

Next, the host system initializes the state of the control unit, caches and mono execution unit by a series of writes to the PPCI port. The last of these specify the start address of the code to execute and tell the processor to start fetching instructions.

Finally, the boot code does any remaining initialization of the processor including any
25 set-up of the PEs (e.g. setting the PE number) before running the application code.

Host services

Once application program is running it will need to access host resources such as the file system.

A protocol is defined between the run-time libraries and a device driver on the host
30 system. This is interrupt based and allows the code running on the MTAP processor to make calls to the application or operating system running on the host.

Debugging

The processor includes hardware support for breakpoints and single-stepping. These are controlled via registers accessed through the PCVI interface.

This interface also allows the debugger, running on the host, to interrogate and update the state of the processor to support fully interactive debugging.

Case Study: Device EV1

To illustrate the use of an MTAP processor in a real device, the architecture of Applicant's evaluation chip EV1 is described in this section. This device is a simple embodiment of Applicant's MTAP processor using Applicant's bus to interface the various ports of the processor to a memory block and to external pins.

EV1 architecture

Figure 9 shows the top level architecture of the EV1 device. The five ports of the MTAP processor and one port of the embedded memory block are connected to a single **ClearConnect** channel (**ClearConnect** is Applicant's bus structure) of six nodes and two lanes in opposing directions. The two external ports at each end of the bus allow the **ClearConnect** bus to be connected from one chip to another. This means a system can be built from multiple EV1 devices to provide the required performance.

The bus ports can also be used to connect to an FPGA to provide other functions, such as peripherals, an external memory controller or a host interface.

The EV1 chip also includes on-chip SRAM which provides fast access to code and data.

MTAP processor

The MTAP processor core in the EV1 has the following specification:

- General
 - Implemented on 0.13 μ process
 - Clock speed: 200 MHz
 - 25 • 4 Kbyte instruction cache: 4-way, 256 lines x 4 instructions, with manual and auto pre-fetch
 - 4 Kbyte data cache, 4-way, 256 lines x 16 bytes
- Mono execution unit
 - 64 byte register file
 - 30 • Support for 8 threads
- Poly execution unit
 - Array of 48 PEs
 - MAC extension to the ALU
 - 4 Kbytes SRAM per PE

- 64 byte register file
- One PIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: 4, 8, 16, 32 bytes per PE
- 5 • Address modes: direct and strided
- One SIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: up to 128 bytes per PE
- Control interfaces
- 10 • Interrupt port: 32-bit address, 32-bit data PPCI
- Register port: 32-bit address, 32-bit data PPCI

This gives a performance of:

- 9,600 MIPS
- 9.6 billion 8x8 MACs / second
- 15 • 38 Gbytes/s memory bandwidth
- 18 Gbytes/s inter-PE bandwidth

Example Applications

Here we give a couple of examples of how the MTAP processor can be applied to specific applications.

20 *Network processing*

In this case, the data to be processed consists of a stream of *packets* of variable size. Each packet consists of header information and a data payload. In its simplest form, the problem is to examine various fields in the header, do some sort of lookup function and route the packet to the appropriate output port. This processing must be
25 done in real time.

In the network processing application, the SIO channels are used for continuously streaming packets into, and out of, PE memory. Within each PE, the software maintains several buffers so that, while one set of packets is being processed, the previous set can be output and, simultaneously, the next set can be loaded.

30 The PIO channels are used by the PEs to send requests to another subsystem that handles lookups. This could on-chip, for example Applicant's Table Lookup Engine, or an off-chip solution such as CAMs.

Bio-informatics

The developing field of *in-silico* drug discovery uses computer simulation rather than the 'wet science' of test tubes to explore the behaviour of potential new drugs.

This includes molecular simulation of individual molecules and the interactions of
5 molecules. For example, the docking of a ligand molecule (drug) into a protein. This simulation requires the calculation of the interaction energies of all atoms of one molecule with all the atoms of the other. This process is repeated for thousands of configurations of the molecules. This results in an embarrassingly parallel problem ideally suited to the MTAP architecture.

10 Each PE is allocated a different configuration of protein and ligand, and performs all of the atom-atom interaction energy calculations. The memory available within a PE is insufficient to hold the details of an entire molecule, however the molecule can be split into pieces and each piece processed in turn. The fetching of atoms can be overlapped with the atom-atom processing and, even with the floating point
15 accelerated MTAP processor, is compute bound. The processing currently performed in most production code is a simplistic model of the interaction of atoms. The processing power that the MTAP architecture makes available allows more sophisticated models to be considered, improving the accuracy and thus value of the results.

20 The MTAP processor is also suited to other bio-informatic tasks such as genetic sequence comparison and pharmacophore fingerprinting.

Software Development Kit

The Software Development Kit (SDK) provides a complete set of development tools: C compiler, assembler linker, debugger, profiler, etc. The software tools use a central
25 configuration file to define the attributes of the target processor that code is being generated for.

Simulation tools

Cycle and bit-accurate simulations of the core are available in C and Verilog. These can be parameterized for the number of PEs, size of memory, and other options. A
30 generic configuration of the C simulator is shipped with the SDK. Once the target system is defined, the simulator can be configured to match that specification.

This allows application development to start before the system architecture is fully defined, and then proceed in parallel with the silicon implementation.

Once the target hardware (or a simulation) is available then the application code can be run and debugged on the target hardware.

Debugger

5 A major challenge when developing code for a parallel architecture is understanding the state of the machine when the code is stopped at a breakpoint or error. The data-parallel nature of the MTAP processor minimizes the difficulty here: a single instruction stream is executing and only the data is different on each PE.

Applicant's 2nd generation debugger builds on previous experience and provides a variety of ways to visualize the state of PE memory and registers. As well as
10 traditional source-level symbolic views of data, and low-level dumps of memory and data, the debugger provides user-defined *picture* views which show the contents of PE memory or registers in graphical form. This provides a quickly understood view of the overall state of the system.

The debugger works identically with all of the simulators and with target hardware. It
15 supports all the features expected from a modern development tool: graphical user interface and command line operation, breakpoints, watchpoints, single-stepping, etc.

Claims

1. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set.
- 5
2. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted
- 10 to operate on said scalar data.
3. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein substantially all said instructions are operable on said parallel and/or said
- 15 scalar data.
4. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of
- 20 processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.
5. A data processor as claimed in claim 1, wherein said scalar part and said
- 25 SIMD part are adapted to operate asynchronously.
6. A data processor as claimed in any of claims 1, 3 or 5, wherein the execution unit comprises a SIMD part comprising an array of processing elements (PEs) adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data.
- 30
7. A data processor as claimed in any of claims 2, 4 or 6, wherein the SIMD part and the scalar part differ only in that each processing element of the SIMD part contains local memory.

8. A data processor as claimed in claim 6, wherein said SIMD part and said scalar part have a common memory area for instructions and scalar data.
9. A data processor as claimed in claim 6, wherein each PE contains an ALU and
5 one or more multiply-accumulate function units.
10. A data processor as claimed in claim 6, wherein each PE contains an ALU and one or more floating point function units.
- 10 11. A data processor as claimed in claim 6, wherein each PE has multiple enable bits to support nested conditional code.
12. A data processor as claimed in claim 6, wherein said PEs and said scalar part each have a similar set of status/result flags which can be used to control branching,
15 on the scalar part of the execution unit, and conditional execution, on the SIMD part of the execution unit, in a similar manner.
13. A data processor as claimed in claim 6, further comprising a communication path between PEs in the SIMD part.
20
14. A data processor as claimed in claim 6, wherein means are provided for data to be transferred to and from the scalar part of the execution unit and the SIMD part of the execution unit.
- 25 15. A data processor as claimed in any of the preceding claims, wherein the control unit is adapted to fetch and execute multiple instruction threads simultaneously.
16. A data processor as claimed in any of the preceding claims, further comprising
30 a semaphore unit adapted to use semaphores to synchronize between instruction threads.
17. A data processor as claimed in claim 16, wherein said semaphores are adapted to synchronize between I/O operations and instruction threads.

SUBSTITUTE SHEET (RULE 26)

18. A data processor as claimed in claim 16, wherein said control unit is adapted to schedule and execute threads based on their priority and the state of any semaphores they are waiting on.

5

19. A data processor as claimed in claim 16, wherein said control unit includes a set of said semaphores adapted to be used to synchronize threads.

20. A data processor as claimed in either of claims 2 or 3, wherein each processing element of the SIMD array comprises one or more function units, a register file, said local memory, and I/O means.

21. A data processor as claimed in any of claims 2, 4, 6 or 7, wherein both said scalar part and said SIMD part have a complete range of addressing modes that allow them to support:

scalar variables pointing to data in scalar memory;
scalar variables pointing to data in memory in the SIMD part of the execution unit;
SIMD variables pointing to memory in the SIMD part of the execution unit; and
SIMD variables pointing to data in scalar memory.

22. A data processor as claimed in any of claims 1, 2 or 5-21, wherein substantially all said instructions are operable on said parallel data or on said scalar data or on a combination of both said parallel data and said scalar data.

25

23. A data processor as claimed in claim 22, further comprising a compiler adapted to produce code from a single source program to operate on both said scalar data and said parallel data.

24. A data processor as claimed in claim 22, wherein said compiler is adapted to detect parallel data automatically.

25. A data processor as claimed in claim 23 or 24, wherein a programmer or said compiler is adapted to use the multi-threaded execution to schedule I/O operations

SUBSTITUTE SHEET (RULE 26)

such that they occur concurrently with computation and data is always available when required.

26. A data processor as claimed in claim 20, wherein said scalar part, said SIMD
5 part and said I/O means are adapted to be operated in parallel by the same instruction stream.

27. A data processor as claimed in claim 26, wherein said I/O means comprises
programmed I/O means adapted to enable each PE to transfer data to and from
10 external scalar memory and wherein each PE provides the address it wants to read data from or write data to.

28. A data processor as claimed in claim 27, wherein memory accesses are
consolidated to minimize the number of memory transfers when multiple PEs are
15 accessing the same memory area.

29. A data processor as claimed in claim 26, wherein each PE is adapted to
transfer data to and from external scalar memory and wherein the address for each PE
is generated automatically based on the PE number and the amount of data being
20 transferred.

30. A data processor as claimed in claim 26, wherein said I/O means comprises
streaming I/O means which distributes an incoming data stream to all the PEs in the
SIMD part and collects data from all the PEs to generate an output stream
25 independently of program execution on the SIMD part of the execution unit.

31. A data processor as claimed in claim 30, wherein the streaming I/O means
only uses enabled PEs to take part in the streaming I/O operation.

30 32. A data processor as claimed in claim 30, wherein the size of data distributed to, and collected from each PE is different for each PE.

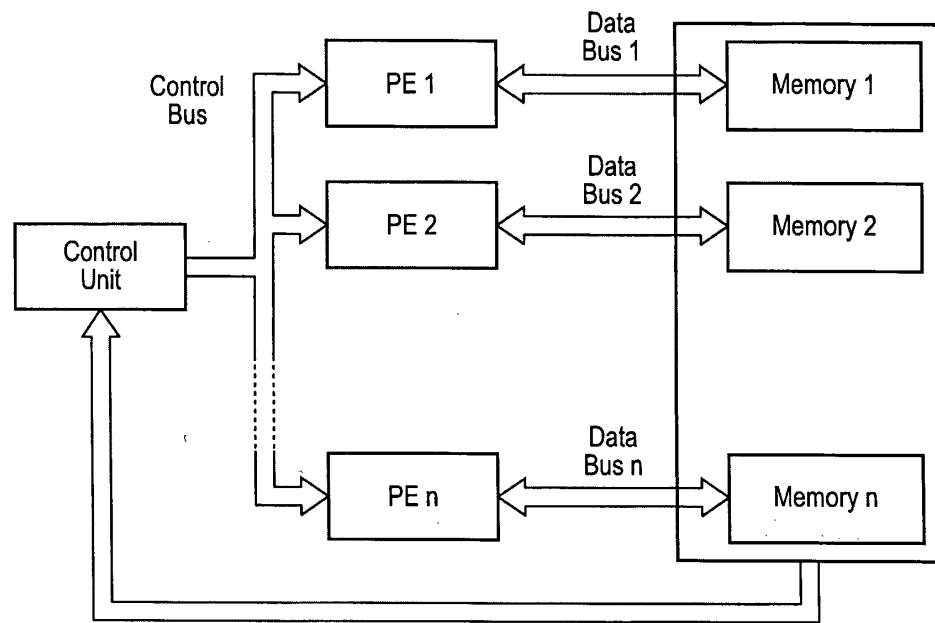


Fig. 1

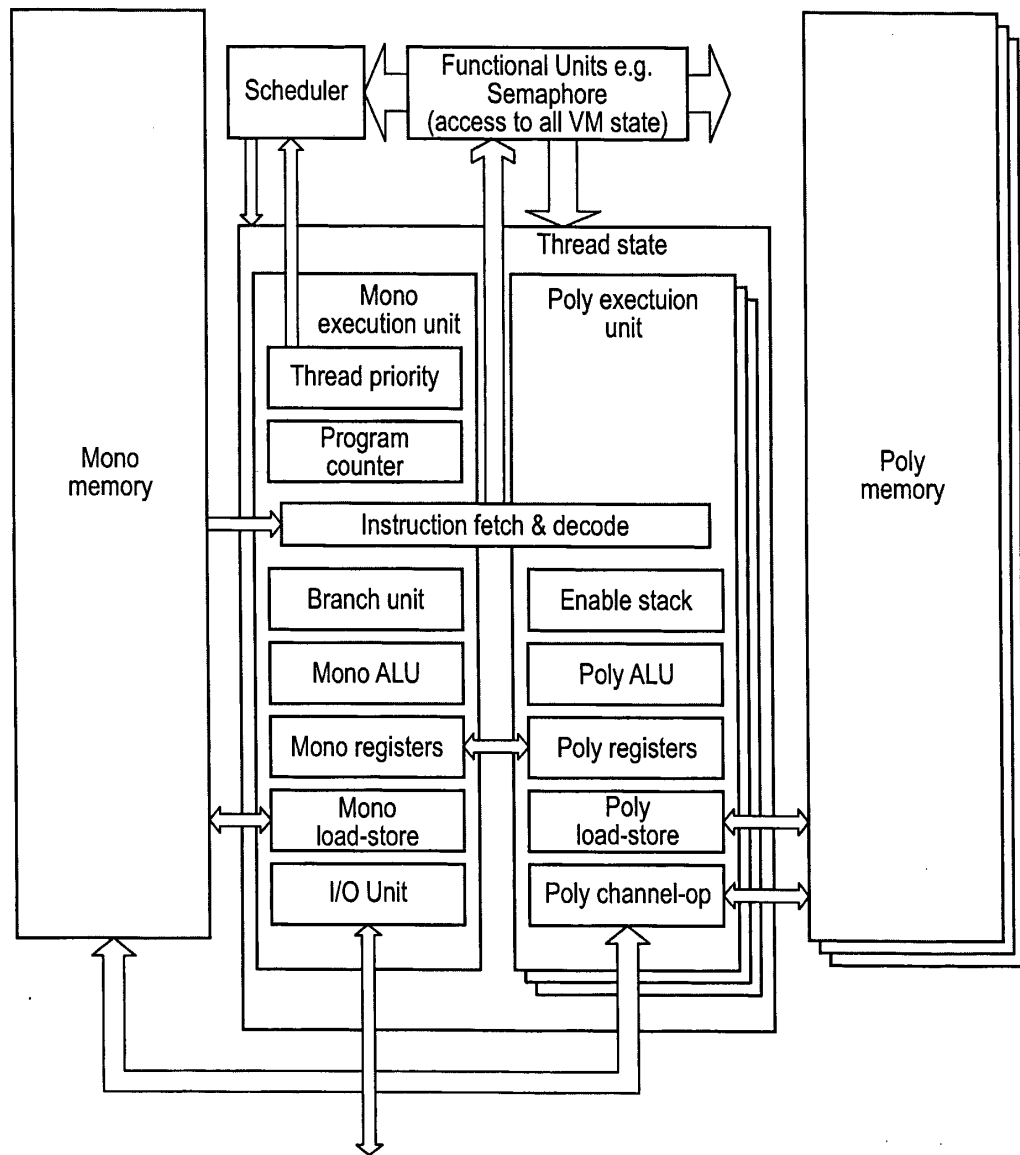


Fig. 2

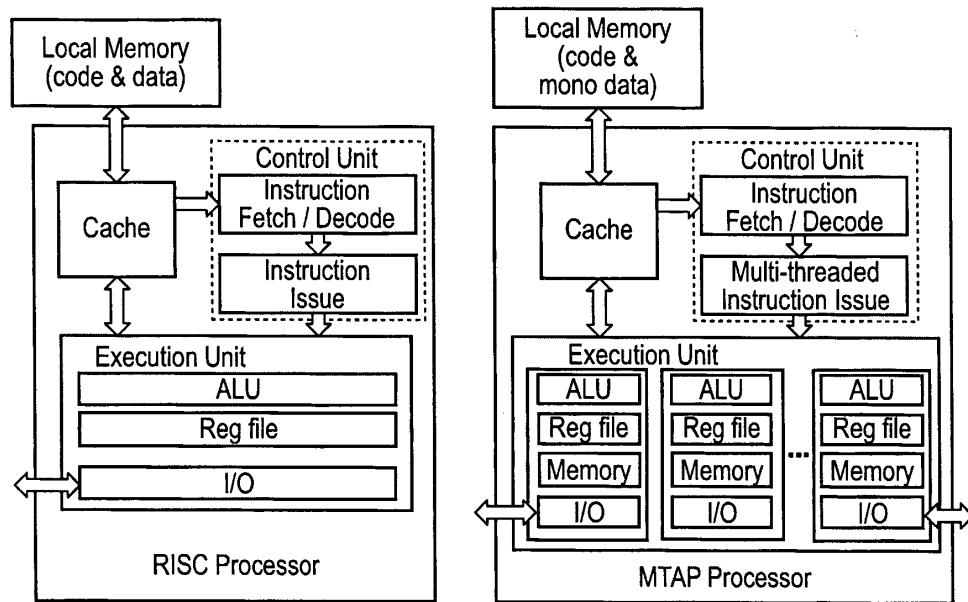


Fig. 3

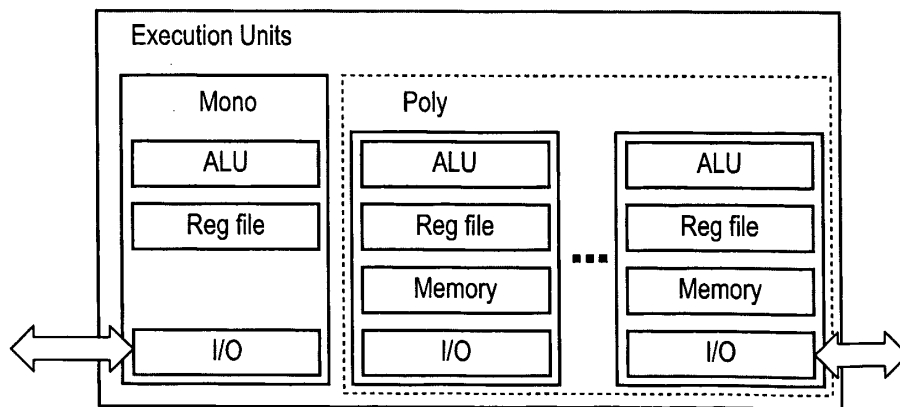


Fig. 4

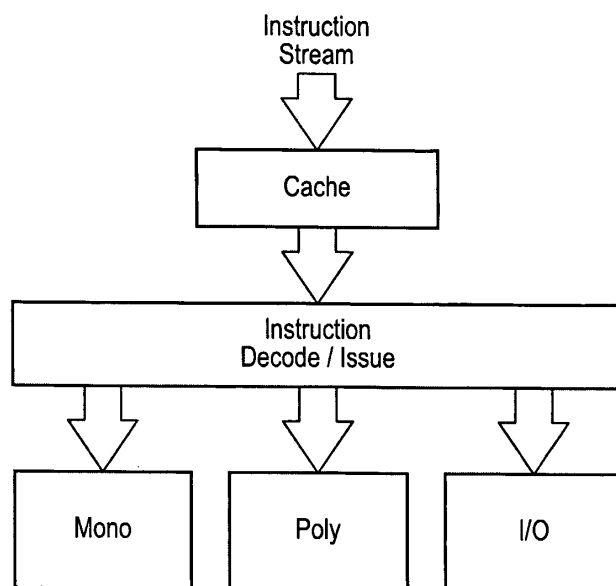


Fig. 5

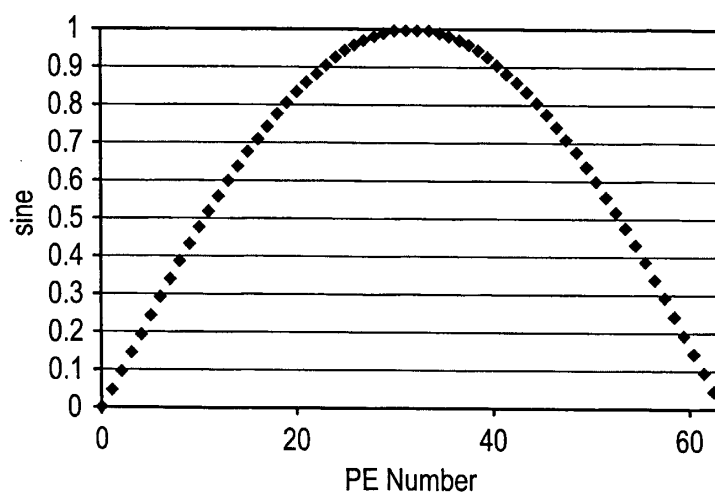


Fig. 6

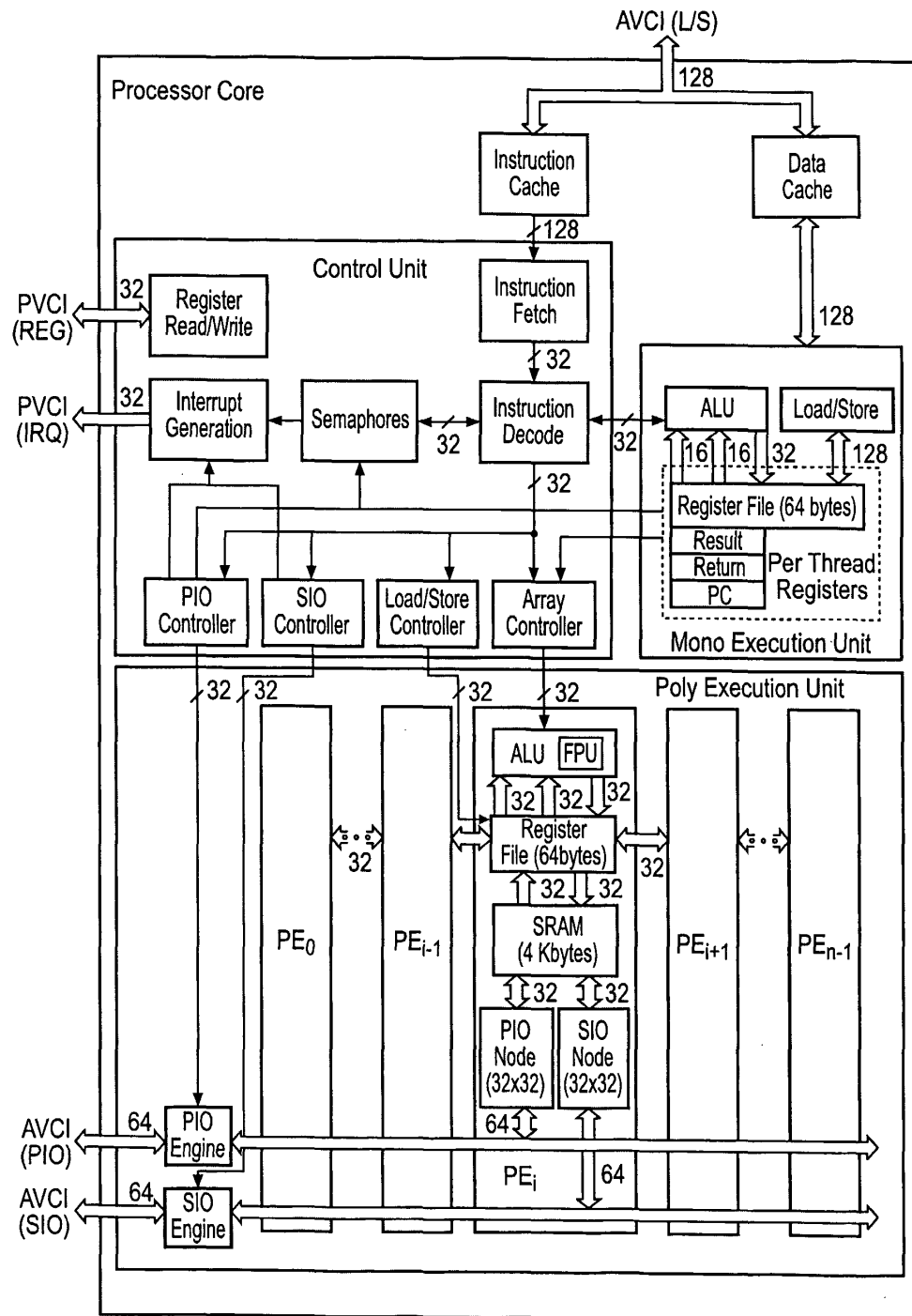


Fig. 7

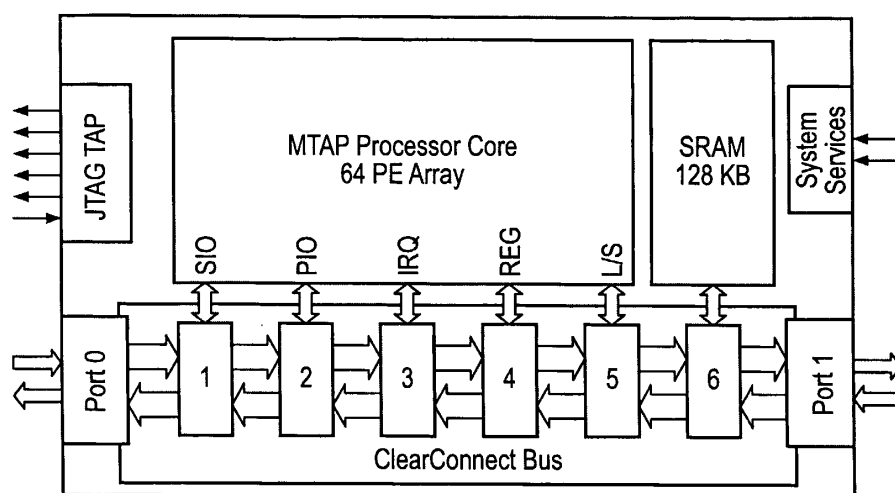


Fig. 8

PATENT COOPERATION TREATY

33

PCT

From the INTERNATIONAL BUREAU

NOTIFICATION OF THE RECORDING
OF A CHANGE(PCT Rule 92bis.1 and
Administrative Instructions, Section 422)

To:

VIGARS, Christopher, Ian
Haseltine Lake
Redcliff Quay
120 Redcliff Street
Bristol BS1 6HU
United KingdomDate of mailing (day/month/year)
01 April 2005 (01.04.2005)Applicant's or agent's file reference
P100390PC00/CIV

IMPORTANT NOTIFICATION

International application No.
PCT/GB2004/004377International filing date (day/month/year)
13 October 2004 (13.10.2004)

1. The following indications appeared on record concerning:

☒ the applicant ☐ the inventor ☐ the agent ☐ the common representative

Name and Address

CLEARSPED TECHNOLOGY LIMITED
3110 Great Western Court
Hunts Ground Road
Stoke Gifford
Bristol BS34 8HP
United Kingdom

State of Nationality

GB

State of Residence

GB

Telephone No.

Facsimile No.

Teleprinter No.

2. The International Bureau hereby notifies the applicant that the following change has been recorded concerning:

☐ the person ☒ the name ☐ the address ☐ the nationality ☐ the residence

Name and Address

CLEARSPED TECHNOLOGY PLC
3110 Great Western Court
Hunts Ground Road
Stoke Gifford
Bristol BS34 8HP
United Kingdom

State of Nationality

GB

State of Residence

GB

Telephone No.

Facsimile No.

Teleprinter No.

EPO -DG 1

07. 04. 2005

3. Further observations, if necessary:

103

4. A copy of this notification has been sent to:

☒ the receiving Office ☐ the designated Offices concerned
☒ the International Searching Authority ☐ the elected Offices concerned
☐ the International Preliminary Examining Authority ☐ other:The International Bureau of WIPO
34, chemin des Colombettes
1211 Geneva 20, Switzerland

Authorized officer

Frederick SAMUELS

Facsimile No. (41-22) 338.89.65

Telephone No. (41-22) 338 9471

12 JANUARY 2005

PATENT COOPERATION TREATY

Appointment of Agent (Power of Attorney)

The undersigned applicant(s) hereby appoint(s):

Benedum, Ulrich Max; **Billington**, Lawrence Emlyn; **Brown**, David Leslie,
Cheyne, John Robert Alexander Mackenzie; **Fenlon**, Christine Lesley; **Gibbs**,
Christopher Stephen; **Giles**, Ashley Simon; **Hitching**, Peter Matthew; **Mohun**, Stephen
John; **Nash**, David Allan; **O'Connell**, David Christopher; **Silverman**, Warren;
Stebbing, Timothy Charles; **Sunderland**, James Harry; **Vigars**, Christopher Ian;
Wilding, Frances Ward; **Williams**, Michael Ian:

all of: **HASELTINE LAKE**
Redcliff Quay, 120 Redcliff Street
Bristol BS1 6HU, ENGLAND

(Telephone: 0117 910 3200; Fax: 0117 910 3201)

as agents

to act on his (her, their) behalf before the competent International Authorities in
connection with the international application concerning (title)¹:

Unified SIMD Processor
(Application No. PCT/GB2004/004377)

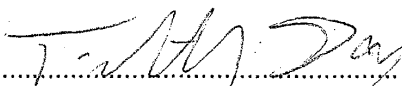
In the name of Clearspeed Technology Limited

[Agents' ref: P100390PC00/CIV]

filed with the United Kingdom or European Patent Office as Receiving Office and to
make or receive payments on their behalf and to sub-authorize further agents so to act.

(Place)² EDINBURGH

(Date)³ 5/1/2005

(Signature)⁴ 

(Name)⁵ DAY, Timothy

(SPECIFIC AUTHORISATION)

see notes on following page

PATENT COOPERATION TREATY

Appointment of Agent (Power of Attorney)

The undersigned applicant(s) hereby appoint(s):

Benedum, Ulrich Max; **Billington**, Lawrence Emlyn; **Brown**, David Leslie,
Cheyne, John Robert Alexander Mackenzie; **Fenlon**, Christine Lesley; **Gibbs**,
Christopher Stephen; **Giles**, Ashley Simon; **Hitching**, Peter Matthew; **Mohun**, Stephen
John; **Nash**, David Allan; **O'Connell**, David Christopher; **Silverman**, Warren;
Stebbing, Timothy Charles; **Sunderland**, James Harry; **Vigars**, Christopher Ian;
Wilding, Frances Ward; **Williams**, Michael Ian:

all of: **HASELTINE LAKE**
Redcliff Quay, 120 Redcliff Street
Bristol BS1 6HU, ENGLAND

(Telephone: 0117 910 3200; Fax: 0117 910 3201)

as agents

to act on his (her, their) behalf before the competent International Authorities in
connection with the international application concerning (title)¹:

Unified SIMD Processor
(Application No. PCT/GB2004/004377)

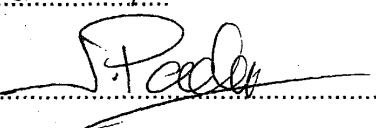
In the name of Clearspeed Technology Limited

[Agents' ref: P100390PC00/CIV]

filed with the United Kingdom or European Patent Office as Receiving Office and to
make or receive payments on their behalf and to sub-authorize further agents so to act.

(Place)² BRISTOL, UK

(Date)³ 4 JAN 2005

(Signature)⁴ 

(Name)⁵ PACKER, James

(SPECIFIC AUTHORISATION)

see notes on following page

12 JANUARY 2005

PATENT COOPERATION TREATY

Appointment of Agent (Power of Attorney)

The undersigned applicant(s) hereby appoint(s):

Benedum, Ulrich Max; **Billington**, Lawrence Emlyn; **Brown**, David Leslie,
Cheyne, John Robert Alexander Mackenzie; **Fenlon**, Christine Lesley; **Gibbs**,
Christopher Stephen; **Giles**, Ashley Simon; **Hitching**, Peter Matthew; **Mohun**, Stephen
John; **Nash**, David Allan; **O'Connell**, David Christopher; **Silverman**, Warren;
Stebbing, Timothy Charles; **Sunderland**, James Harry; **Vigars**, Christopher Ian;
Wilding, Frances Ward; **Williams**, Michael Ian:

all of: **HASELTINE LAKE**
Redcliff Quay, 120 Redcliff Street
Bristol BS1 6HU, ENGLAND

(Telephone: 0117 910 3200; Fax: 0117 910 3201)

as agents

to act on his (her, their) behalf before the competent International Authorities in
connection with the international application concerning (title)¹:

Unified SIMD Processor
(Application No. PCT/GB2004/004377)

In the name of Clearspeed Technology Limited

[Agents' ref: P100390PC00/CIV]

filed with the United Kingdom or European Patent Office as Receiving Office and to
make or receive payments on their behalf and to sub-authorize further agents so to act.

(Place)² BRISTOL U.K.

(Date)³ 4 JANUARY 2005

(Signature)⁴ 

(Name)⁵ WILLIAMS, David

(SPECIFIC AUTHORISATION)

see notes on following page

PCT/GB 2004 / 0 0 4 3 7 7
12 JANUARY 2005

PATENT COOPERATION TREATY

Appointment of Agent (Power of Attorney)

The undersigned applicant(s) hereby appoint(s):

Benedum, Ulrich Max; **Billington**, Lawrence Emlyn; **Brown**, David Leslie,
Cheyne, John Robert Alexander Mackenzie; **Fenlon**, Christine Lesley; **Gibbs**,
Christopher Stephen; **Giles**, Ashley Simon; **Hitching**, Peter Matthew; **Mohun**, Stephen
John; **Nash**, David Allan; **O'Connell**, David Christopher; **Silverman**, Warren;
Stebbing, Timothy Charles; **Sunderland**, James Harry; **Vigars**, Christopher Ian;
Wilding, Frances Ward; **Williams**, Michael Ian:

all of: **HASELTINE LAKE**
Redcliff Quay, 120 Redcliff Street
Bristol BS1 6HU, ENGLAND

(Telephone: 0117 910 3200; Fax: 0117 910 3201)

as agents

to act on his (her, their) behalf before the competent International Authorities in
connection with the international application concerning (title)¹:

Unified SIMD Processor
(Application No. PCT/GB2004/004377)

In the name of Clearspeed Technology Limited

[Agents' ref: P100390PC00/CIV]

filed with the United Kingdom or European Patent Office as Receiving Office and to
make or receive payments on their behalf and to sub-authorize further agents so to act.

(Place)² BRISTOL, UK

(Date)³ 4/1/05

(Signature)⁴ Colin S. Davidson

(Name)⁵ DAVIDSON, Colin

(SPECIFIC AUTHORISATION)

see notes on following page

12 JANUARY 2005

PATENT COOPERATION TREATY

Appointment of Agent (Power of Attorney)

The undersigned applicant(s) hereby appoint(s):

Benedum, Ulrich Max; **Billington**, Lawrence Emllyn; **Brown**, David Leslie,
Cheyne, John Robert Alexander Mackenzie; **Fenlon**, Christine Lesley; **Gibbs**,
Christopher Stephen; **Giles**, Ashley Simon; **Hitching**, Peter Matthew; **Mohun**, Stephen
John; **Nash**, David Allan; **O'Connell**, David Christopher; **Silverman**, Warren;
Stebbing, Timothy Charles; **Sunderland**, James Harry; **Vigars**, Christopher Ian;
Wilding, Frances Ward; **Williams**, Michael Ian:

all of: **HASELTINE LAKE**
Redcliff Quay, 120 Redcliff Street
Bristol BS1 6HU, ENGLAND

(Telephone: 0117 910 3200; Fax: 0117 910 3201)

as agents

to act on his (her, their) behalf before the competent International Authorities in
connection with the international application concerning (title)¹:

Unified SIMD Processor
(Application No. PCT/GB2004/004377)

In the name of Clearspeed Technology Limited

[Agents' ref: P100390PC00/CIV]

filed with the United Kingdom or European Patent Office as Receiving Office and to
make or receive payments on their behalf and to sub-authorize further agents so to act.

(Place)² BRISTOL UK.....

(Date)³ 4/1/2005.....

(Signature)⁴ Neil Hickey.....

(Name)⁵ HICKEY, Neil

(SPECIFIC AUTHORISATION)

see notes on following page

12 JANUARY 2005

PATENT COOPERATION TREATY

Appointment of Agent (Power of Attorney)

The undersigned applicant(s) hereby appoint(s):

Benedum, Ulrich Max; **Billington**, Lawrence Emlyn; **Brown**, David Leslie,
Cheyne, John Robert Alexander Mackenzie; **Fenlon**, Christine Lesley; **Gibbs**,
Christopher Stephen; **Giles**, Ashley Simon; **Hitching**, Peter Matthew; **Mohun**, Stephen
John; **Nash**, David Allan; **O'Connell**, David Christopher; **Silverman**, Warren;
Stebbing, Timothy Charles; **Sunderland**, James Harry; **Vigars**, Christopher Ian;
Wilding, Frances Ward; **Williams**, Michael Ian:

all of: **HASELTINE LAKE**
Redcliff Quay, 120 Redcliff Street
Bristol BS1 6HU, ENGLAND

(Telephone: 0117 910 3200; Fax: 0117 910 3201)

as agents

to act on his (her, their) behalf before the competent International Authorities in
connection with the international application concerning (title)¹:

Unified SIMD Processor
(Application No. PCT/GB2004/004377)

In the name of Clearspeed Technology Limited

[Agents' ref: P100390PC00/CIV]

filed with the United Kingdom or European Patent Office as Receiving Office and to
make or receive payments on their behalf and to sub-authorize further agents so to act.

(Place)² Bristol, UK

(Date)³ 4th January, 2005

(Signature)⁴ 

(Name)⁵ **STUTTARD**, David

(SPECIFIC AUTHORISATION)

see notes on following page

Abstract**Unified SIMD Processor**

5 A data processor comprises a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set. The execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data. Substantially all the instructions are operable on the parallel and/or the
10 scalar data. Preferably, both the scalar part and the SIMD part have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the execution unit; SIMD variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory. The scalar part, the
15 SIMD part, and I/O means are adapted to be operated in parallel by the same instruction stream.

[Figure 2]

Claims

1. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set.
5
2. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted
10 to operate on said scalar data.
3. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein substantially all said instructions are operable on said parallel and/or said
15 scalar data.
4. A data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of
20 processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.
5. A data processor as claimed in claim 1, wherein said scalar part and said
25 SIMD part are adapted to operate asynchronously.
6. A data processor as claimed in any of claims 1, 3 or 5, wherein the execution unit comprises a SIMD part comprising an array of processing elements (PEs) adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data.
30
7. A data processor as claimed in any of claims 2, 4 or 6, wherein the SIMD part and the scalar part differ only in that each processing element of the SIMD part contains local memory.

8. A data processor as claimed in claim 6, wherein said SIMD part and said scalar part have a common memory area for instructions and scalar data.
9. A data processor as claimed in claim 6, wherein each PE contains an ALU and
5 one or more multiply-accumulate function units.
10. A data processor as claimed in claim 6, wherein each PE contains an ALU and one or more floating point function units.
- 10 11. A data processor as claimed in claim 6, wherein each PE has multiple enable bits to support nested conditional code.
12. A data processor as claimed in claim 6, wherein said PEs and said scalar part each have a similar set of status/result flags which can be used to control branching,
15 on the scalar part of the execution unit, and conditional execution, on the SIMD part of the execution unit, in a similar manner.
13. A data processor as claimed in claim 6, further comprising a communication path between PEs in the SIMD part.
20
14. A data processor as claimed in claim 6, wherein means are provided for data to be transferred to and from the scalar part of the execution unit and the SIMD part of the execution unit.
- 25 15. A data processor as claimed in any of the preceding claims, wherein the control unit is adapted to fetch and execute multiple instruction threads simultaneously.
16. A data processor as claimed in any of the preceding claims, further comprising
30 a semaphore unit adapted to use semaphores to synchronize between instruction threads.
17. A data processor as claimed in claim 16, wherein said semaphores are adapted to synchronize between I/O operations and instruction threads.

18. A data processor as claimed in claim 16, wherein said control unit is adapted to schedule and execute threads based on their priority and the state of any semaphores they are waiting on.

5

19. A data processor as claimed in claim 16, wherein said control unit includes a set of said semaphores adapted to be used to synchronize threads.

20. A data processor as claimed in either of claims 2 or 3, wherein each
10 processing element of the SIMD array comprises one or more function units, a register file, said local memory, and I/O means.

21. A data processor as claimed in any of claims 2, 4, 6 or 7, wherein both said scalar part and said SIMD part have a complete range of addressing modes that allow
15 them to support:

scalar variables pointing to data in scalar memory;

scalar variables pointing to data in memory in the SIMD part of the execution unit;

SIMD variables pointing to memory in the SIMD part of the execution unit; and

20 SIMD variables pointing to data in scalar memory.

22. A data processor as claimed in any of claims 1, 2 or 5-21, wherein substantially all said instructions are operable on said parallel data or on said scalar data or on a combination of both said parallel data and said scalar data.

25

23. A data processor as claimed in claim 22, further comprising a compiler adapted to produce code from a single source program to operate on both said scalar data and said parallel data.

30 24. A data processor as claimed in claim 22, wherein said compiler is adapted to detect parallel data automatically.

25. A data processor as claimed in claim 23 or 24, wherein a programmer or said compiler is adapted to use the multi-threaded execution to schedule I/O operations

such that they occur concurrently with computation and data is always available when required.

26. A data processor as claimed in claim 20, wherein said scalar part, said SIMD
5 part and said I/O means are adapted to be operated in parallel by the same instruction stream.

27. A data processor as claimed in claim 26, wherein said I/O means comprises
programmed I/O means adapted to enable each PE to transfer data to and from
10 external scalar memory and wherein each PE provides the address it wants to read data from or write data to.

28. A data processor as claimed in claim 27, wherein memory accesses are
consolidated to minimize the number of memory transfers when multiple PEs are
15 accessing the same memory area.

29. A data processor as claimed in claim 26, wherein each PE is adapted to
transfer data to and from external scalar memory and wherein the address for each PE
is generated automatically based on the PE number and the amount of data being
20 transferred.

30. A data processor as claimed in claim 26, wherein said I/O means comprises
streaming I/O means which distributes an incoming data stream to all the PEs in the
SIMD part and collects data from all the PEs to generate an output stream
25 independently of program execution on the SIMD part of the execution unit.

31. A data processor as claimed in claim 30, wherein the streaming I/O means
only uses enabled PEs to take part in the streaming I/O operation.

30 32. A data processor as claimed in claim 30, wherein the size of data distributed to, and collected from each PE is different for each PE.

Unified SIMD Processor

Background to the Invention

The invention consists of a significant improvement to a well known type of parallel processor. Single-Instruction Multiple-Data (SIMD) architectures are widely used in a number of computing applications. They have the advantage of high performance, low power consumption and easy scalability. A SIMD processor differs from the more commonly used multi-processor system (where a number of processors, each independently executing their own program code, are used together to solve a problem) in that it executes a single stream of instructions but each instruction operates on multiple data items in parallel.

There are a range of SIMD processor architectures. These range from an *intelligent memory* type of approach where each *column* of a memory array has a simple 1-bit Arithmetic Logic Unit (ALU) through to more complex architectures where a, usually smaller, array of more complex processing elements are fed with a single instruction stream. They have, however, been difficult to program in the past.

In all these cases the array of SIMD processing elements are fed with instructions and data by a dedicated controller. Often the controller is the largest and most complex part of the design because of the need to match the data processing rate achieved by the SIMD array. The array controller is not itself 'intelligent' and is, in turn, controlled by a general purpose processor which also runs the bulk of the application software. This *host* processor will execute most of the application code; it will send data to be processed to the SIMD array with an indication of the function to be performed, typically a pointer to the start of a sequence of SIMD array instructions or microinstructions. An exemplary prior art SIMD processor is outlined in Figure 1.

This introduces two programming problems. Firstly, the array controller can only be programmed at a very low level: a series of microcode, or possibly assembler, instructions to be executed by the array. This is because the array controller is not designed to work with a compiler. Secondly, this programming is completely separated from the main application code running on the host processor. This may be 'wrapped up' by providing C functions or C++ classes which makes them appear more integrated and reduces the complexity of passing control between the host processor and the SIMD array. Also, in the simplest form, the host processor and the SIMD array will have completely separate memory spaces for code and data. Any of

the standard shared memory techniques can be used to address this with all the resulting problems: complex arbitration, cache coherency issues, contention for memory bandwidth, inter-processor synchronization, greater cost, etc.

5 A preferable solution is to provide a fully integrated architecture where the same processor can transparently process both *scalar* and SIMD operations in a fully integrated fashion. This means that a single program can be written, in a standard high-level language such as C, which includes all parts of the application – both ‘normal’ (scalar) data and parallel data processing.

Prior Art

10 The basic SIMD processing model has been in use for many years. Some of the earliest examples are the ILLIAC IV developed in 1972, which had 64 processors with floating point and memory, and Goodyear’s STARAN (1975) which was a 1-bit architecture. Many variants have been developed since then.

15 There is also a recent trend to put small-scale SIMD extensions in standard processor architectures. This is exemplified by the Intel MMX and SSE extensions to the x86 architecture. This allows, for example, the values in two 32-bit registers to be added as if each register contained four independent 8-bit values. This is not considered relevant to the current discussion for a number of reasons:

- the SIMD processing is done on a very small scale;
- 20 • only a limited range of SIMD functionality is provided;
- there are distinct instructions for operating on SIMD data;
- the same hardware is used for SIMD and for scalar operations (although separate registers may be provided for the SIMD operands)

25 The idea of providing indexed addressing modes in a SIMD processor is disclosed in our copending patent application GB9908225.7. This invention extends that idea by using a more flexible implementation of that idea in combination with a more powerful array controller to provide a unified processor architecture.

Summary of the Invention

30 Applicant’s approach, after many years of experience with developing systems based on SIMD coprocessors, was to create a new architecture where instructions are decoded and despatched to the appropriate part of the execution unit: either the scalar (mono) execution unit or the SIMD (poly) execution unit. Instruction execution on these are decoupled so that, for example, a mono instruction followed by a poly instruction will execute concurrently. The support for mono and poly data is, as far as

possible, identical – this makes it practical to write a compiler to target the processor. This greatly simplifies the task of programming the processor.

This means a single program runs on the processor which does both mono and poly operations as required – typically it will do both at the same time. A single C program
5 can be written which manages program control flow, operates on mono data, and operates on poly data.

To support this, and further simplify programming, this unity of the architecture is reflected in the instruction set and programming model. The processing elements in the SIMD array and the mono execution unit have a common instruction set. For
10 instance, an **add** instruction can be used with operands which are either poly registers or mono registers (or immediate values); the appropriate execution unit will execute the instruction. Mono and poly operands can also be mixed to allow, for example, a single mono value to be added to a poly register which holds a different value on every processing element.

15 It is appropriate to note that this does not require the mono execution unit to have exactly the same micro-architecture as the poly execution unit. For example, the poly processing elements could have an 8-bit ALU, while the mono execution unit has a 32-bit ALU.

A block schematic diagram of a processor in accordance with the invention is shown
20 in Figure 2.

The most significant features contributing to the novelty and inventiveness of this invention

include the following concepts:

- The processor provides a unified programming model: a single program runs on
25 a processor providing both scalar and SIMD operations.
- This is supported with a unified instruction set: e.g. an **add** instruction can work with either mono or poly operands (or a mixture of the two)
- Poly operations can freely use mono registers in expressions, and more importantly for addressing memory.
- 30 • Having a full range of addressing modes and completely separate memory blocks in each PE make it simple to support poly data structures and poly pointers on the SIMD array (previous SIMD architectures either provide a single address to all PEs or use sequences of instructions to construct an address in memory).

- Mono and poly operations are of equal standing: wherever possible, all instructions can be used with either mono or poly operands (there are some exceptions because a small set of operations, mainly for control, only make sense for one or the other).
 - 5 • The mono and poly execution units are loosely coupled so that their operation can be overlapped. Compilers are extremely good at scheduling instructions to take advantage of this flexibility - maximizing the efficiency of the system.
 - There are some compiler optimizations specific to this architecture: for example it may be better, in some instances, to move an operation that would normally
10 be done on the mono execution unit on to the poly execution unit - either because it can then be overlapped with other operations on the mono execution unit or because the result is needed on the poly execution unit anyway. Again, the compiler can detect such cases and make the appropriate optimizations.
 - A common code and data space is available to both mono and poly operations
15 making it easy to generate code for, and share data between, the two execution units.
 - The overall architecture is suitable for targeting by a high level language compiler - again for both mono and poly operations.
- 20 To this end, the invention provides, in a first aspect a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set.
- According to a second aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel
25 and scalar data under the same instruction set, and wherein the execution unit comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data.
- According to a third aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel
30 and scalar data under the same instruction set, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.
- According to a fourth aspect, the invention provides a data processor comprising a single control unit and an integral execution unit adapted to operate on both parallel and scalar data under the same instruction set, and wherein the execution unit

comprises a SIMD part comprising an array of processing elements adapted to operate on said parallel data and a scalar part adapted to operate on said scalar data, and wherein substantially all said instructions are operable on said parallel and/or said scalar data.

- 5 The single control unit and the integral execution unit may be adapted to operate asynchronously on both the parallel and scalar data under the same instruction set. The execution unit preferably comprises a SIMD part comprising an array of processing elements adapted to operate on the parallel data and a scalar part adapted to operate on the scalar data.

- 10 Preferably, the SIMD part and the scalar part differ only in that each processing element of the array contains local memory. The SIMD part and the scalar part preferably have a common memory area for instructions and scalar data. Each PE may contain an ALU and one or more multiply-accumulate function units or an ALU and one or more floating point function units. Preferably, each PE has multiple
15 enable bits to support nested conditional code.

The SIMD and the scalar execution unit conveniently have a similar set of status/result flags which can be used to control branching on the scalar part of the execution unit and conditional execution on the SIMD part of the execution unit in a similar manner.

- 20 There is preferably a communication path between PEs in the SIMD part and means are preferably provided for data to be transferred to and from the scalar part of the execution unit and the SIMD part of the execution unit.

- Advantageously the control unit is adapted to fetch and execute multiple instruction threads simultaneously. A semaphore unit may use semaphores to synchronize
25 between instruction threads. The semaphores may be used to synchronize between I/O operations and instruction threads. The control unit may be adapted to schedule and execute threads based on their priority and the state of any semaphores they are waiting on and may include a set of the semaphores for use in synchronizing threads. Each processing element of the SIMD array may comprise one or more function units,
30 a register file, said local memory, and I/O means.

Both the scalar part and the SIMD part preferably have a complete range of addressing modes that allow them to support: scalar variables pointing to data in scalar memory; scalar variables pointing to data in memory in the SIMD part of the

execution unit; SIMD variables pointing to memory in the SIMD part of the execution unit; and SIMD variables pointing to data in scalar memory.

Substantially all the instructions are preferably operable on either the parallel data or the scalar data or a combination of both the scalar data and the parallel data.

- 5 The processor may further comprise a compiler adapted to produce code from a single source program to operate on both the scalar and the parallel data. The compiler is preferably adapted to detect parallel data automatically.

The scalar part, the SIMD part, and the I/O means are preferably adapted to be operated in parallel by the same instruction stream.

- 10 A programmer or the compiler is adapted to use the multi-threaded execution to schedule I/O operations such that they occur concurrently with computation and data is always available when required.

The I/O means may comprise programmed I/O means adapted to enable each PE to transfer data to and from external scalar memory and wherein each PE provides the

- 15 address it wants to read data from or write data to. The memory accesses may be consolidated to minimize the number of memory transfers when multiple PEs are accessing the same memory area.

Each PE is preferably adapted to transfer data to and from external scalar memory, in which case the address for each PE is generated automatically based on the PE

- 20 number and the amount of data being transferred.

The I/O means may comprise streaming I/O means which distributes an incoming data stream to all the PEs in the SIMD part and collects data from all the PEs to generate an output stream independently of program execution on the SIMD part of the execution unit. The streaming I/O means may only use enabled PEs to take part in

25 the streaming I/O operation. The size of data distributed to, and collected from each PE may be different for each PE.

Brief Description of the Drawings

The invention will be described with reference to the following drawings, in which:

- Figure 1 shows a prior art SIMD processor;
- 30 Figure 2 shows the architecture in accordance with the present invention;
- Figure 3 is a comparison between RISC and MTAP architecture;
- Figure 4 is a schematic diagram of an MTAP execution unit;
- Figure 5 indicates schematically the execution of an instruction;
- Figure 6 is a graph indicating the power efficiency of the MTAP architecture;

Figure 7 is a block schematic diagram depicting an exemplary MTAP processor; and

Figure 8 is a block diagram of a specific implementation of the MTAP architecture as applied to an evaluation chip.

5 Detailed Description of the Illustrated Embodiments

The design of this processor architecture was driven by the desire to make it practical and efficient to program in a high level language. In order to make this possible a number of features are required. First, the overall programming model needs to be simple and 'regular' – i.e. the same operations and addressing modes can be applied to all data types.

The execution unit for mono data is based on well known principles used in many RISC processors. This makes it simple for a compiler to target and easy for a user to understand.

The functionality supported by each PE is then made as similar as possible to the mono execution unit: each PE has an ALU, a register file and memory, and supports a range of addressing modes for transferring data between memory and registers.

It is not necessary for the PEs to be identical to the mono execution unit in every detail: e.g. the number of registers and the width and complexity of the ALU are likely to be different in practice. The poly execution unit gets its performance largely from the number of PEs brought to bear on the problem rather than the processing power of the individual PEs. To balance this, the mono execution unit is likely to require a wider and more complex ALU. This is reasonable because this is only instantiated once while the PEs are replicated many times and need to be efficiently implemented.

The number of clock cycles required to execute an instruction on the mono and poly execution units is highly variable. Both will execute some instructions in a single cycle. But some instructions could take several cycles on the mono execution unit or on the poly execution unit. It is therefore essential that the two execution units are only loosely coupled rather than operating in lock-step; this allows them to overlap their execution of instructions. The hardware will synchronize their operations when necessary: e.g. when they need to access a shared resource, or one execution unit requires data from the other.

Transfers between the registers and memory, and other I/O transfers, can take place concurrently with ALU operations. To simplify the task for the compiler, these

interactions are all interlocked using register scoreboarding (see our copending patent application GB 9908203.4).

An important aspect of the PE architecture to support compiled code is the existence of a full set of addressing modes: direct, indirect and indexed. Indirect and indexed
5 addressing by the PEs is described in the previously mentioned copending patent application GB 9908225.7.

Reference may also be had to our copending patent application GB 0321186.9 for details of the "ClearConnect" bus referred to in the context of the present invention.

In a high level language compiler for this architecture, it is only necessary to
10 introduce a simple modification; the keyword 'poly' used in a declaration identifies data that is distributed across the SIMD array. Similar methods have been used before, for example the 'shape' keyword used in C*, a language used for programming the "Connection Machine". However, it is only with the present architecture that poly variables become 'first class' objects – i.e. they can be used everywhere that 'normal'
15 mono data can be: in expressions, as function arguments, in conditional statements, etc. Pointers to both mono and poly data are allowed, as are pointers of both mono and poly types. Furthermore, poly variables can be mixed quite freely with mono data in all these contexts. This provides a very simple and regular programming model for the user.

20 Note that it would also be possible for a compiler to detect the parallelism in the program or data automatically: This is frequently done with 'vectorizing' Fortran compilers used with supercomputers, for example. The same techniques could be applied to this architecture. The architecture will now be described in more detail.

The Multi-Threaded Array Processing (MTAP) architecture has been developed to
25 address a number of problems in high performance, high data rate processing.

A number of applications are characterized by very high data rates, flexible processing requirements and hard real-time constraints. We use the term *data flow processing* to describe this class of problem.

The MTAP processor delivers on the three primary requirements for data flow
30 applications:

1. It directly addresses the data bandwidth, and has a clear scalability path for future requirements.

2. It provides the raw horsepower for the processing functions required, on the maximum data rate that the system will encounter. That processing power can scale with increasing function demands.
3. It stores the data close to the processing core to maximize bandwidth and minimize latency.

MTAP Overview

The MTAP architecture defines a family of embedded processors with parallel data processing capability. Figure 3 compares a standard processor and the MTAP architecture. As can be seen, the MTAP processor has a standard, RISC-like, control unit with instruction fetch, caches and I/O mechanisms. This is coupled to a highly parallel execution unit which provides the performance and scalability of the MTAP architecture.

- The processor is designed in a highly modular fashion which allows many details of a specific implementation to be easily customized for the target application.
- 15 To simplify the integration of the processor into a variety of systems, the processor can be also configured to be big or little-endian.

Control unit

The control unit fetches, decodes and dispatches instructions to the execution units. The processor executes a fairly standard, three operand instruction set. The manner in which the same instruction stream is decoded and issued in parallel to the mono part, the poly part and the I/O part is schematically indicated in Figure 5.

- The control unit also provides hardware support for multi-threaded execution, allowing fast swapping between multiple threads. The threads are prioritized and are intended primarily to support efficient overlap of I/O and compute. This can be used to hide the latency of external data accesses.

The MTAP processor can also include instruction and data caches to minimize the latency of memory accesses. The size and type of these is a configurable option. Alternatively, the caches may be replaced with SRAM to provide a complete embedded solution.

- 30 The control unit also includes a control port which is used for initializing and debugging the processor; it includes support for breakpoints, single stepping and the examination of internal state.

Execution unit

The execution unit consists of a number of *processing elements* (PEs). This allows it to process data elements in parallel. Each PE consists of an ALU, registers, memory and I/O.

- 5 The number of PEs in a processor core is a configurable parameter which allows performance to be scaled to meet the needs of the application (see Figure 6). The execution unit can be thought of as two, largely independent, parts (see Figure 4).

One PE forms the *mono* execution unit; this is dedicated to processing mono (i.e. scalar or non-parallel) data. The mono execution unit also handles program flow control such as branching and thread switching. The rest of the PEs form the *poly* execution unit which processes parallel (poly) data.

The poly execution unit may consist of tens, hundreds or even thousands of PEs. This array of PEs operates in a synchronous manner, similar to SIMD, where every PE executes the same instruction on its piece of data. Each PE also has its own independent local memory; this provides fast access to the data being processed. For example, one PE at 400 MHz has a memory bandwidth of 1.6 Gbytes/s. An array with 256 such PEs has an aggregate bandwidth of over 400 Gbytes/s, with single cycle latency. The number of registers and the amount of memory in each PE are configurable options.

Input / output

The MTAP processor core has two basic I/O mechanisms. The first, *Programmed I/O* (PIO), is the normal mechanism used for accessing memory external to the MTAP core: it supports random accesses to variable sized data by the PEs. The second mechanism is *Streaming I/O* (SIO) which allows chunks of contiguous data to be streamed directly into the memory within PEs.

Each of these supports a variety of addressing modes, described in more detail later.

Programming model

From a programmer's perspective, the MTAP processor appears as a single processor running a single C program. This is very different from some other parallel processing models where the programmer has to explicitly program multiple independent processors, or can only access the processor via function calls or some other indirect mechanism.

The MTAP processor executes a single instruction stream; each instruction is sent to one of the functional units: this may be the mono or poly execution unit or one of the

I/O controllers. The processor can despatch an instruction on every cycle.

For multi-cycle instructions, the operation of the functional units can be overlapped.

So, for example, an I/O operation can be started with one instruction and on the next cycle the mono execution unit could start a multiply instruction (which requires
5 several cycles to execute). While these operations are proceeding, the poly execution unit can continue to execute instructions.

The main change from programming a standard processor is the concept of operating on parallel data. Data to be processed is assigned to variables which have an instance on every PE and are operated on in parallel: we call this *poly* data. This can be
10 thought of as a data vector which is distributed across the array.

Variables only requiring a single instance (e.g. loop control variables) are known as *mono* variables – they behave exactly like normal variables on a sequential processor.

Applicant provides a compiler which uses a simple extension to standard C to identify data which is to be processed in parallel. The new keyword **poly** is used in a
15 declaration to define data which exists, and is processed, on every PE in the poly execution unit.

Example

To give a feel for the way the MTAP processor is programmed, the following represents a fragment of code which will calculate 64 values of a sine function across
20 the PE array in a single operation.

```
#include <fnext.h>
#include <math.h>
#define PI 3.1415926535897932384
#define NUMBER_OF_PES 64
25
poly float angle, sine;
poly int pe;

/* get PE number: 0...n-1 */
30 pe = get_penum();
/* convert to an angle in range 0 to Pi */
angle = pe * PI / NUMBER_OF_PES;
/* calculate sine of angle on each PE */
sine = sinp(angle);
```

This code uses the library function `get_penum()` to get a unique value in the variable `pe` on each PE. This is scaled to give a range of values for `angle` between 0 and π across the PEs.

Finally, the library function `sinp()` is called to calculate the sine of these values on all
5 PEs simultaneously. The `sinp` function is the poly equivalent of the standard sine function; it takes a poly argument and returns the appropriate value on every PE.

Performance

The performance of an MTAP core with 64 PEs, on a 0.13 μ process and running at 400 MHz is shown in Table 1. The columns show the performance achieved with the
10 basic PE, the PE with an additional Multiply-Accumulate (MAC) unit, and the PE with an FPU extension.

Table 1 - Performance Comparison

	Base Architecture	8-bit MAC	FPU
MIPS	25,600	25,600	25,600
MMACS ($\times 10^6$)	—	25,600	25,600
MFLOPS	—	—	51,200
Memory Bandwidth (Gbytes / s)	100	100	100
Inter-PE Bandwidth (GBytes / s)	50	50	50
Relative Area	1.0	1.1	1.3

15 However, the core can be scaled well beyond this number of PEs, to 256 or even thousands of PEs. It also scales down efficiently to lower numbers of PEs, or lower clock rates, enabling low cost and low power systems to be built.

Figure 6 shows how performance and power scale with the number of PEs. By choosing the number of PEs and the clock frequency, the designer has a great deal of
20 flexibility to tune performance, bandwidth, cost and power for the needs of a given application.

Architecture Details

The following sections describe the MTAP processor in more detail. A block diagram of the processor is shown in Figure 7. This is a conceptual view, which reflects the programming model. In practice, the control and mono execution units form a tightly-coupled pipeline which provides overall control of the processor.

Interfaces

The processor has a number of external interfaces. The widths of each of these are configurable. All these interfaces currently use the Virtual Component Interface Standard defined by the VSI Alliance.

- 10 There are three categories of interface:

Mono data & instructions

This is a single *Advanced* VCI (AVCI) interface which is used for mono loads and stores, and for instruction fetching.

Poly data

- 15 This uses one or more AVCI interfaces for PIO and SIO data. The number of physical interfaces corresponds to the number of PIO and SIO channels implemented.

Control

- There are two control interfaces which use the *Peripheral* VCI (PVCi) protocol. One is used for initialization and debug. The other allows the MTAP processor to generate
20 interrupts to the host system.

Instruction set

- The processor has a fairly standard RISC-like instruction set. Most instructions can operate on mono or poly operands and are executed by the appropriate part of the execution unit. Some instructions are only relevant to either the mono or poly
25 execution unit, for example all program flow control is handled by the mono unit.

The instruction set provides a standard set of functions on both mono and poly execution units:

- Integer and floating point adds, subtracts, multiplies, divides
- Logical operations: and, or, not, xor
- 30 • Arithmetic and logical shifts
- Data comparisons: equal, not equal, greater than, less than, etc.
- Data movement between registers
- Loads and stores between registers and memory

To give a feel for the nature of assembly code for the MTAP processor, a few lines of code are shown below. This simple example loads a value from memory into a mono register, gets the PE number into a register on each PE and then adds these two values together producing a different result on every PE.

```
5  ld 0:m4, 0x3000    // load mono reg 0 from mem
   penum 8:p4         // PE number into poly reg
   add 4:p4, 0:m4, 8:m4 // add; result in reg 4
```

Control unit

The control unit fetches instructions from memory, decodes them and despatches them to the appropriate functional unit.

The controller includes a scheduler to provide hardware support for multi-threaded code. This is a vital part of the architecture for achieving the performance potential of the MTAP processor. Because of the highly parallel architecture of the poly execution unit, there can be significant latencies if all PEs need to read or write external data.

When part of an application stalls because it is waiting for data from external memory, the processor can switch to another code thread that is ready to run. This serves to hide the latency of accesses and keep the processor busy.

The number of threads is a configurable option. The threads are prioritized: the processor will run the highest priority thread that is ready to run; a higher priority thread can pre-empt a lower priority thread when it becomes ready to run. Threads are synchronized – with each other and with hardware, such as I/O engines – via semaphores.

In the simplest case of multi-threaded code, a program would have two threads: one for I/O and one for compute. By *pre-fetching* data in the I/O thread, the programmer (or the compiler) can ensure the data is available when it is required by the execution units and the processor can run without stalling.

Execution units

This section describes the common aspects of the poly and mono execution units.

ALU operations

Instructions for arithmetic and logical operations are provided in several versions:

- Various sizes: 1, 2, 3 and 4 bytes, and combinations of sizes
- Various data types: signed and unsigned integers, and floating-point
- Mixes of signed/unsigned operations

Status register

Associated with the ALU is a status register; this contains five status bits that provide information about the result of the last ALU operation. When set, these bits indicate:

- Most significant bit set
- 5 • Carry generated
- Overflow generated
- Negative result
- Zero result

Registers

- 10 To support operations on data of different widths, the registers in the PEs can be accessed very flexibly. The register files are best thought of as an array of bytes which can be addressed as registers of 1 to 8 bytes wide.

The mono and poly registers are addressed in a consistent way using byte addresses and widths specified in bytes. The mono register file is 16 bits wide and so all
15 addresses and widths must be a multiple of 2. There are no alignment restrictions on poly register accesses.

Addressing modes

- Load and store instructions are used to transfer data between memory and registers. In the case of the mono execution unit, these transfer data to and from memory external
20 to the MTAP processor.

Poly loads and stores transfer data between the PE register file and PE memory. Data is transferred between the PEs and external memory using the I/O functions described later.

- There are three addressing modes for loads and stores which can be used for both
25 mono and poly data. These are:

Direct

The address to be read/written is specified as an immediate value.

Indirect

The address is specified in a register.

- 30 **Indexed**

The address is calculated from adding an offset to a base address in a register. The offset must be an immediate value.

Conditional code

The main difference between code for the mono and the poly execution units is the handling of conditional execution.

The mono unit uses conditional jumps to branch around code, typically based on the result of the previous instructions. This means that mono conditions affect both mono
5 and poly operations. This is just like a standard RISC architecture.

The poly unit uses a set of enable bits (described in more detail below) to control whether each PE will have its state changed instructions it executes. This provides per-PE predicated operation. The enable state can be changed based on the result of a
10 previous operation.

The following sections describe the architectural features of the two execution units in more detail.

Mono execution unit

The mono execution unit is a 16-bit processing element consisting of:

- 15 • A 16-bit ALU with optional extensions
- A register file of configurable size
- Status and control registers

The mono ALU extensions include a multiplier, a barrel shifter and a normalizer, which is important for accelerating software implementations of floating point
20 operations.

As well as handling mono data, the mono unit is responsible for program flow control (branching), thread switching and other control functions. The mono execution unit also has overall control of I/O operations of poly data. Results from these operations are returned to a register in the mono unit.

25 Conditional execution

The mono execution unit handles conditional execution in the same way as a traditional processor. A set of conditional and unconditional jump instructions use the result of previous operations to jump over conditional code, back to the start of loops, etc.

30 Multi-threaded execution

The MTAP processor supports several hardware threads. There is a hardware scheduler in the control unit and the mono execution unit maintains multiple banks of critical registers for fast context switching.

The threads are prioritized (0 being highest priority). Control of execution between the threads is performed using semaphores under programmer control. Higher priority threads will only yield to lower priority threads when they are stalled on yielding instructions (such as semaphore wait operations). Lower priority threads can be pre-empted at any time by higher priority ones.

Semaphores are special registers that can be incremented or decremented with atomic (non-interruptible) operations called *signal* and *wait*. A *signal* instruction will increment a semaphore. A *wait* will decrement a semaphore unless the semaphore is 0, in which case it will stall until the semaphore is signalled by another thread.

Semaphores can also be accessed by hardware units (such as the I/O controllers) to synchronize these with software.

Poly execution unit

The poly execution unit is an array of Processing Elements (PEs). Each PE in the poly execution unit consists of:

- An 8-bit ALU with optional extensions such as a multiply-accumulate (MAC) unit
- A register file of configurable size
- Status and enable registers
- A block of memory of configurable size
- An inter-PE communication path
- One or more I/O channels

Load and store instructions move data between a PE's register file and memory, while the ALU operates on data in the register file. Data is transferred in to, and out of, the PE's memory using I/O instructions.

ALU

The poly ALU is used for performing arithmetic and logical operations on values held in the PE register file.

While the ALU is only 8 bits wide, instructions exist for multi-byte arithmetic which is handled by iteration. ALU extensions exist to accelerate functions for floating point, DSP, etc. For example, an optional integer multiply-accumulate unit (MAC) can be included in the ALU. This can deliver an 8 x 8 bit MAC result every cycle, or a 16 x 16 bit MAC every four cycles. The accumulated result can be up to 64 bits wide. The standard ALU includes basic hardware support to accelerate floating point operations.

Conditional behaviour

The SIMD nature of the PE array prohibits each PE having its own branch unit (branching being handled by the mono execution unit). Instead, each PE can control whether its state should be updated by the current instruction by *enabling* or *disabling* itself; this is rather like the predicated instructions in some RISC CPUs.

Enable state

A PE's enable state is determined by a number of bits in the enable register. If all these bits are set to one, then a PE is *enabled* and executes instructions normally. If one or more of the enable bits is zero, then the PE is *disabled* and most instructions it receives will be ignored (instructions on the enable state itself, for example, are not be disabled).

The enable register is treated as a stack, and new bits can be pushed onto the top of the stack allowing nested predicated execution. The result of a test, either a 1 or a 0, is pushed onto the enable stack. This bit can later be popped from the top of the stack to remove the effect of that condition. This makes handling nested conditions and loops very efficient. Note that, although the enable stack is of fixed size, the compiler handles saving and restoring the state automatically, so there are no limitations on compiled code. When programming at the assembler level, it is the programmer's responsibility to manage the stack.

Instructions

Conditional execution on the poly execution unit is supported by a set of poly conditional instructions: **if**, **else**, **endif**, etc. These manage the enable bits to allow different PEs to execute each branch of an **if...else** construct in C, for example. These also support nested conditions by pushing and popping the condition value on the enable stack.

As a simple example, consider the following code fragment:

```
// disable PEs where reg 32 is non-zero
if.eq 32:p1, 0 // push result onto stack
// increment reg 8 on enabled PEs
30 add 8:p4, 8:p4, 1
// return all PEs to original enable state
endif // pop enable stack
```

Here, the initial **if** instruction compares the two operands on each PE. If they are equal it pushes 1 onto the top of the enable stack – this leaves those PEs enabled if they

were previously enabled and disabled if they were already disabled. If the two operands are not equal, a 0 is pushed onto the stack – this disables the corresponding PEs.

The following **add** instruction is sent to all PEs, but only acted on by those that are still enabled. Finally, the **endif** instruction pops the enable stack, returning all PEs to their original enable state.

Forced loads and stores

Poly loads and stores are normally predicated by the enable state of the PE. However, because there are instances where it is necessary to load and store data regardless of the current enable state, the instruction set includes *forced* loads and stores. These will change the state of the PE even if it is disabled.

I/O mechanisms

There are two I/O mechanisms provided for transferring data between PE memory and devices outside the MTAP core. *Programmed I/O* (PIO) extends the load/ store model: it is used for transfers of small amounts of data between PE memory and external memory. *Streaming I/O* (SIO) is used to efficiently transfer contiguous chunks of data in and out of PE memory.

The number of PIO and SIO channels in a processor is a configurable parameter. A processor will always have at least one PIO channel as this is required for normal program execution.

Multiple I/O channels can run simultaneously.

I/O architecture

The I/O systems consist of three parts: Controller, Engine and Node.

Controller

The PIO and SIO controllers decode I/O instructions and coordinate with the rest of the control unit and the mono processor. The controllers synchronize with software threads via semaphores.

Engine

The I/O engines are basically DMA engines which manage the actual data transfer.

There is a Controller and Engine for each I/O channel. A single Controller can manage several I/O Engines.

Node

There is an I/O Node in each PE. The I/O Engine activates each Node in turn allowing to serialize the data transfers. The Nodes provide buffering of data to minimize the impact of I/O on the performance of the PEs.

5 **Programmed I/O (PIO)**

PIO is closely coupled to program execution and is the normal way for the processor to transfer data to and from the outside world (e.g. external memory, hardware accelerators or a host processor).

The PIO mechanism provides a number of addressing modes:

10 **Direct addressed**

Each PE provides an external memory address for its data. This provides random access to data.

Strided

15 The external memory address is incremented for each PE. In each case, the size of data transferred to each PE is the same.

When multiple PEs are accessing memory then the transfers can be *consolidated* so as to perform the minimum number of external accesses. So, for example, if half the processors are reading one location and the other half reading another, then only two memory reads would be performed. In fact, consolidation can be better than that:
20 because the bus transfers are *packetized*, even transfers from nearby addresses can be effectively consolidated.

Streaming I/O (SIO)

SIO is used for streaming high-bandwidth data directly to and from PE memory. This is less flexible but very efficient for transferring blocks of data in and out of the
25 system. This is typically used to stream data to and from memory mapped I/O devices. Each PE can transfer a different size block of data.

Streaming I/O can be made very efficient in a number of ways. For example, multi-threaded code allows data I/O and compute to be fully overlapped. Also, multiple data buffers can be allocated in PE memory so that different chunks of data can be input,
30 processed and streamed out simultaneously.

Swazzle

Finally, the PEs are able to communicate with one another via what is known as the *swazzle* path that connects the register file of each PE with the register files of its left and right neighbours. On each cycle, PE_n can perform a register-to-register transfer of

16 bits to either its left or right neighbour, PE_{n-1} or PE_{n+1} , while simultaneously receiving data from the other neighbour.

Swizzle instructions use multiples of 2 bytes in their arguments, so the source and destination registers must be 2-byte aligned. Instructions are provided to shift data left or right through the array, and to swap data between adjacent PEs.

The enable state of a PE affects its participation in a swizzle operation in the following way: if a PE is enabled, then its registers may be updated by a neighbouring PE, regardless of the neighbouring PE's enable state. Conversely, if a PE is disabled, its register file will not be altered by a neighbour under any circumstance. A disabled PE will still provide data to an enabled neighbour.

The data written into the registers of the PEs at the ends of the swizzle path can be set by the mono execution unit.

Host interface

The interfaces to the host system are used for 3 basic purposes: initialization and booting, access to host services and debugging.

Initialization

There are a number of stages of initialization required to start code running on the MTAP processor. These are normally handled transparently by the development tools, but an overview is provided here as background information.

First, the application code (including bootstrap) is loaded into memory.

Next, the host system initializes the state of the control unit, caches and mono execution unit by a series of writes to the PSCI port. The last of these specify the start address of the code to execute and tell the processor to start fetching instructions.

Finally, the boot code does any remaining initialization of the processor including any set-up of the PEs (e.g. setting the PE number) before running the application code.

Host services

Once application program is running it will need to access host resources such as the file system.

A protocol is defined between the run-time libraries and a device driver on the host system. This is interrupt based and allows the code running on the MTAP processor to make calls to the application or operating system running on the host.

Debugging

The processor includes hardware support for breakpoints and single-stepping. These are controlled via registers accessed through the PSCI interface.

This interface also allows the debugger, running on the host, to interrogate and update the state of the processor to support fully interactive debugging.

Case Study: Device EV1

To illustrate the use of an MTAP processor in a real device, the architecture of Applicant's evaluation chip EV1 is described in this section. This device is a simple embodiment of Applicant's MTAP processor using Applicant's bus to interface the various ports of the processor to a memory block and to external pins.

EV1 architecture

Figure 9 shows the top level architecture of the EV1 device. The five ports of the MTAP processor and one port of the embedded memory block are connected to a single **ClearConnect** channel (**ClearConnect** is Applicant's bus structure) of six nodes and two lanes in opposing directions. The two external ports at each end of the bus allow the **ClearConnect** bus to be connected from one chip to another. This means a system can be built from multiple EV1 devices to provide the required performance.

The bus ports can also be used to connect to an FPGA to provide other functions, such as peripherals, an external memory controller or a host interface.

The EV1 chip also includes on-chip SRAM which provides fast access to code and data.

MTAP processor

The MTAP processor core in the EV1 has the following specification:

- General
 - Implemented on 0.13 μ process
 - Clock speed: 200 MHz
 - 4 Kbyte instruction cache: 4-way, 256 lines x 4 instructions, with manual and auto pre-fetch
 - 4 Kbyte data cache, 4-way, 256 lines x 16 bytes
- Mono execution unit
 - 64 byte register file
 - Support for 8 threads
- Poly execution unit
 - Array of 48 PEs
 - MAC extension to the ALU
 - 4 Kbytes SRAM per PE

- 64 byte register file
- One PIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: 4, 8, 16, 32 bytes per PE
- 5 • Address modes: direct and strided
- One SIO channel
 - AVCI port: 32-bit address, 64-bit data
 - Transfer size: up to 128 bytes per PE
- Control interfaces
- 10 • Interrupt port: 32-bit address, 32-bit data PPCI
- Register port: 32-bit address, 32-bit data PPCI

This gives a performance of:

- 9,600 MIPS
- 9.6 billion 8x8 MACs / second
- 15 • 38 Gbytes/s memory bandwidth
- 18 Gbytes/s inter-PE bandwidth

Example Applications

Here we give a couple of examples of how the MTAP processor can be applied to specific applications.

20 **Network processing**

In this case, the data to be processed consists of a stream of *packets* of variable size. Each packet consists of header information and a data payload. In its simplest form, the problem is to examine various fields in the header, do some sort of lookup function and route the packet to the appropriate output port. This processing must be
25 done in real time.

In the network processing application, the SIO channels are used for continuously streaming packets into, and out of, PE memory. Within each PE, the software maintains several buffers so that, while one set of packets is being processed, the previous set can be output and, simultaneously, the next set can be loaded.

30 The PIO channels are used by the PEs to send requests to another subsystem that handles lookups. This could on-chip, for example Applicant's Table Lookup Engine, or an off-chip solution such as CAMs.

Bio-informatics

The developing field of *in-silico* drug discovery uses computer simulation rather than the 'wet science' of test tubes to explore the behaviour of potential new drugs.

- This includes molecular simulation of individual molecules and the interactions of
5 molecules. For example, the docking of a ligand molecule (drug) into a protein. This simulation requires the calculation of the interaction energies of all atoms of one molecule with all the atoms of the other. This process is repeated for thousands of configurations of the molecules. This results in an embarrassingly parallel problem ideally suited to the MTAP architecture.
- 10 Each PE is allocated a different configuration of protein and ligand, and performs all of the atom-atom interaction energy calculations. The memory available within a PE is insufficient to hold the details of an entire molecule, however the molecule can be split into pieces and each piece processed in turn. The fetching of atoms can be overlapped with the atom-atom processing and, even with the floating point
15 accelerated MTAP processor, is compute bound. The processing currently performed in most production code is a simplistic model of the interaction of atoms. The processing power that the MTAP architecture makes available allows more sophisticated models to be considered, improving the accuracy and thus value of the results.
- 20 The MTAP processor is also suited to other bio-informatic tasks such as genetic sequence comparison and pharmacophore fingerprinting.

Software Development Kit

- The Software Development Kit (SDK) provides a complete set of development tools: C compiler, assembler linker, debugger, profiler, etc. The software tools use a central
25 configuration file to define the attributes of the target processor that code is being generated for.

Simulation tools

- Cycle and bit-accurate simulations of the core are available in C and Verilog. These can be parameterized for the number of PEs, size of memory, and other options. A
30 generic configuration of the C simulator is shipped with the SDK. Once the target system is defined, the simulator can be configured to match that specification.
- This allows application development to start before the system architecture is fully defined, and then proceed in parallel with the silicon implementation.

Once the target hardware (or a simulation) is available then the application code can be run and debugged on the target hardware.

Debugger

5 A major challenge when developing code for a parallel architecture is understanding the state of the machine when the code is stopped at a breakpoint or error. The data-parallel nature of the MTAP processor minimizes the difficulty here: a single instruction stream is executing and only the data is different on each PE.

Applicant's 2nd generation debugger builds on previous experience and provides a variety of ways to visualize the state of PE memory and registers. As well as
10 traditional source-level symbolic views of data, and low-level dumps of memory and data, the debugger provides user-defined *picture* views which show the contents of PE memory or registers in graphical form. This provides a quickly understood view of the overall state of the system.

The debugger works identically with all of the simulators and with target hardware. It
15 supports all the features expected from a modern development tool: graphical user interface and command line operation, breakpoints, watchpoints, single-stepping, etc.

1/6

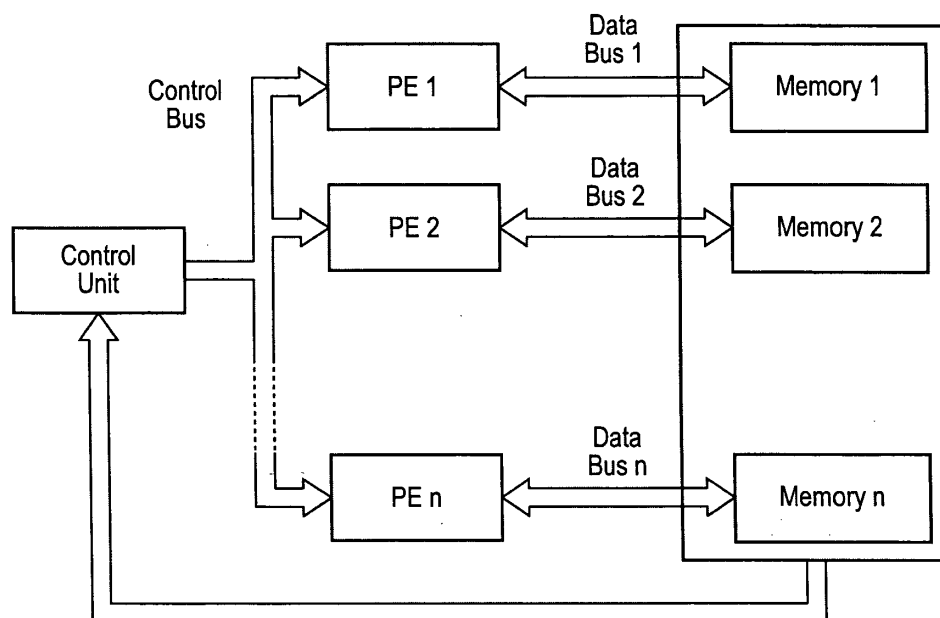


Fig. 1

2/6

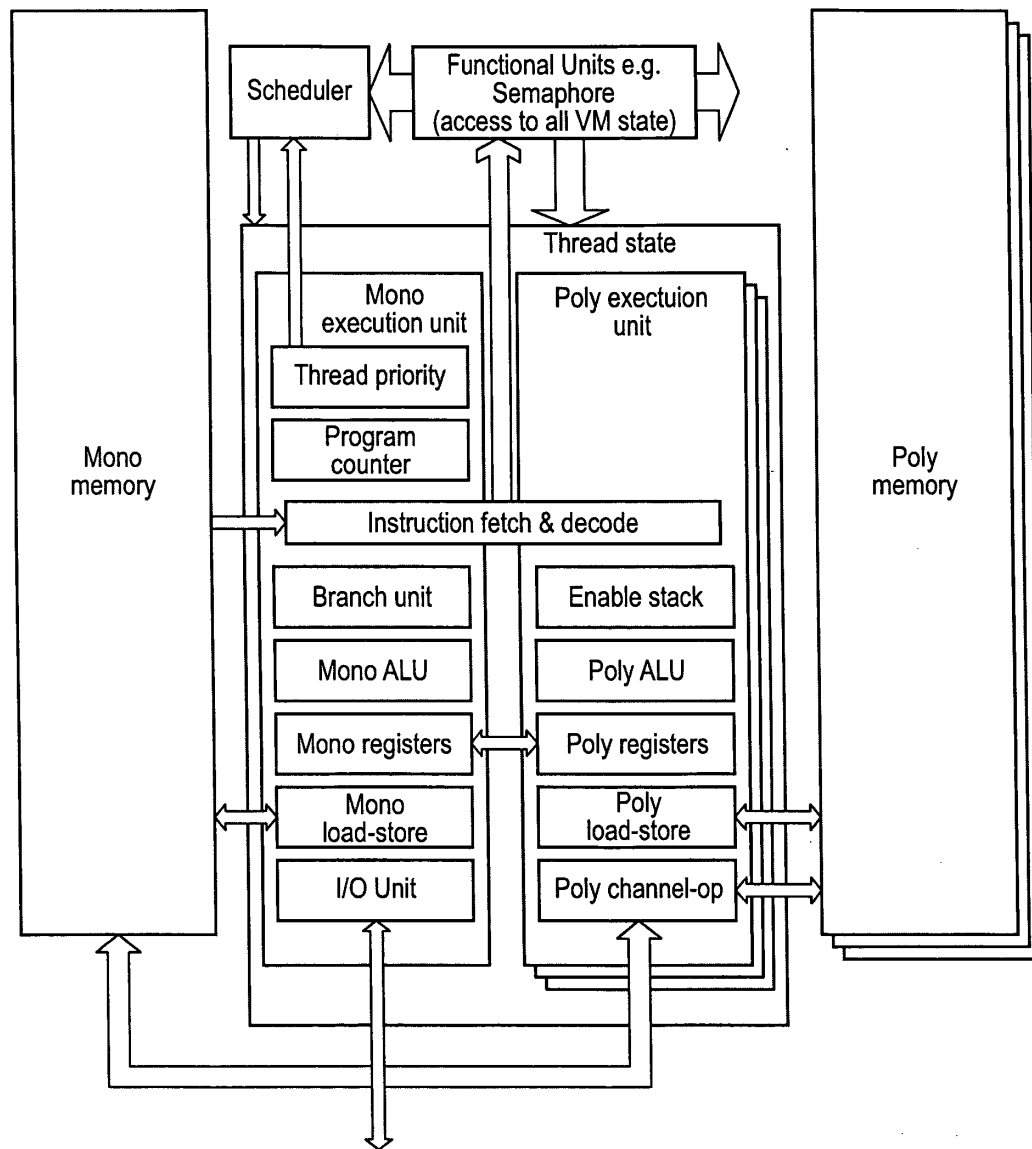


Fig. 2

3/6

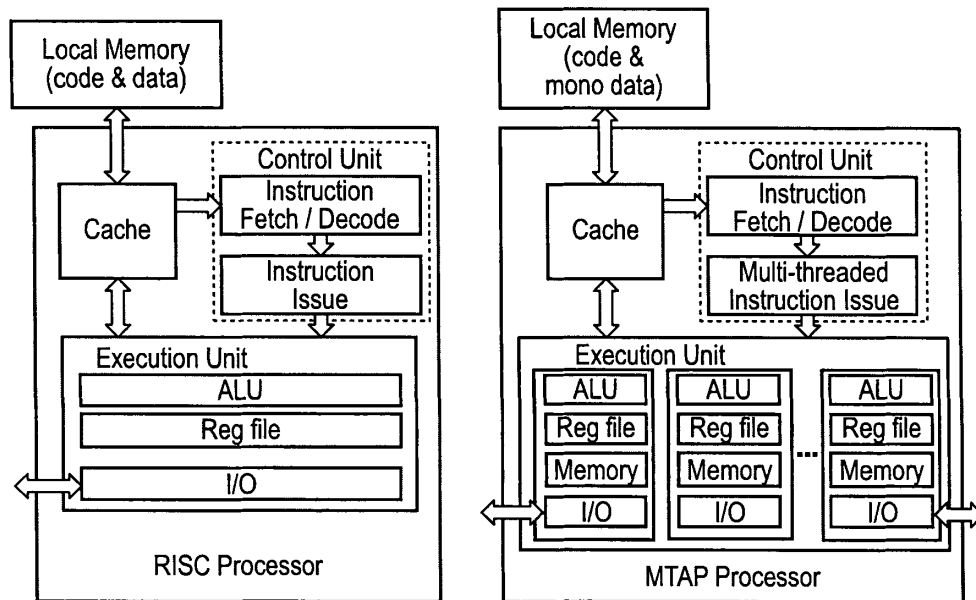


Fig. 3

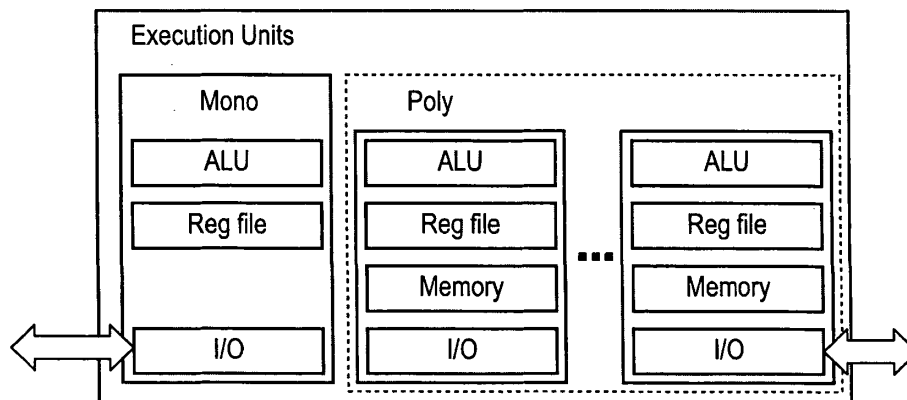


Fig. 4

4/6

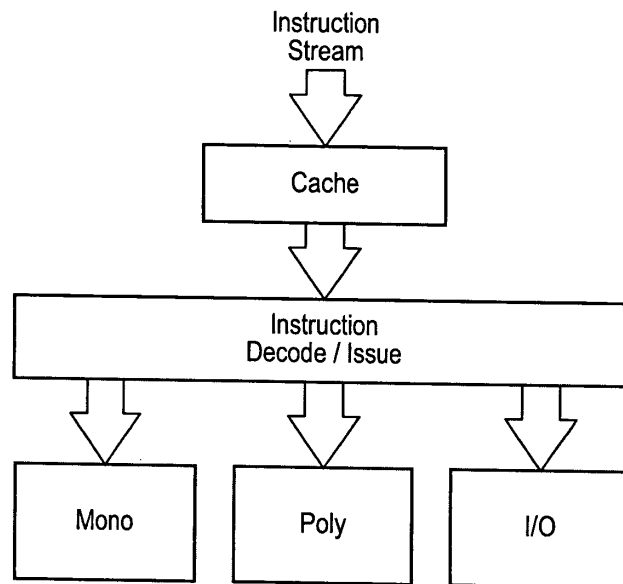


Fig. 5

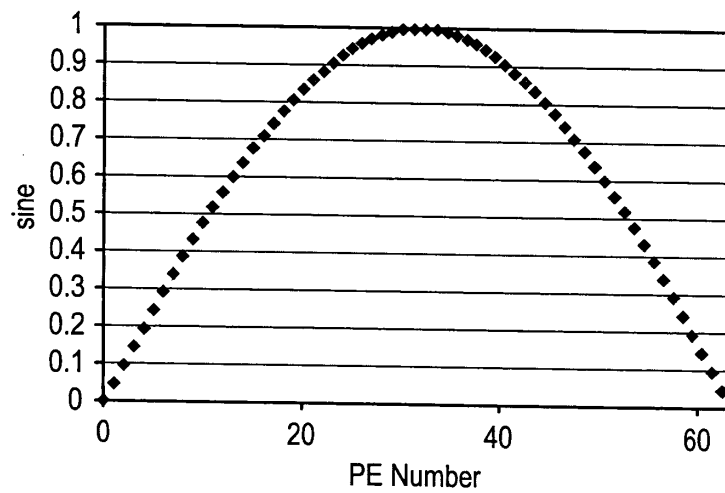


Fig. 6

5/6

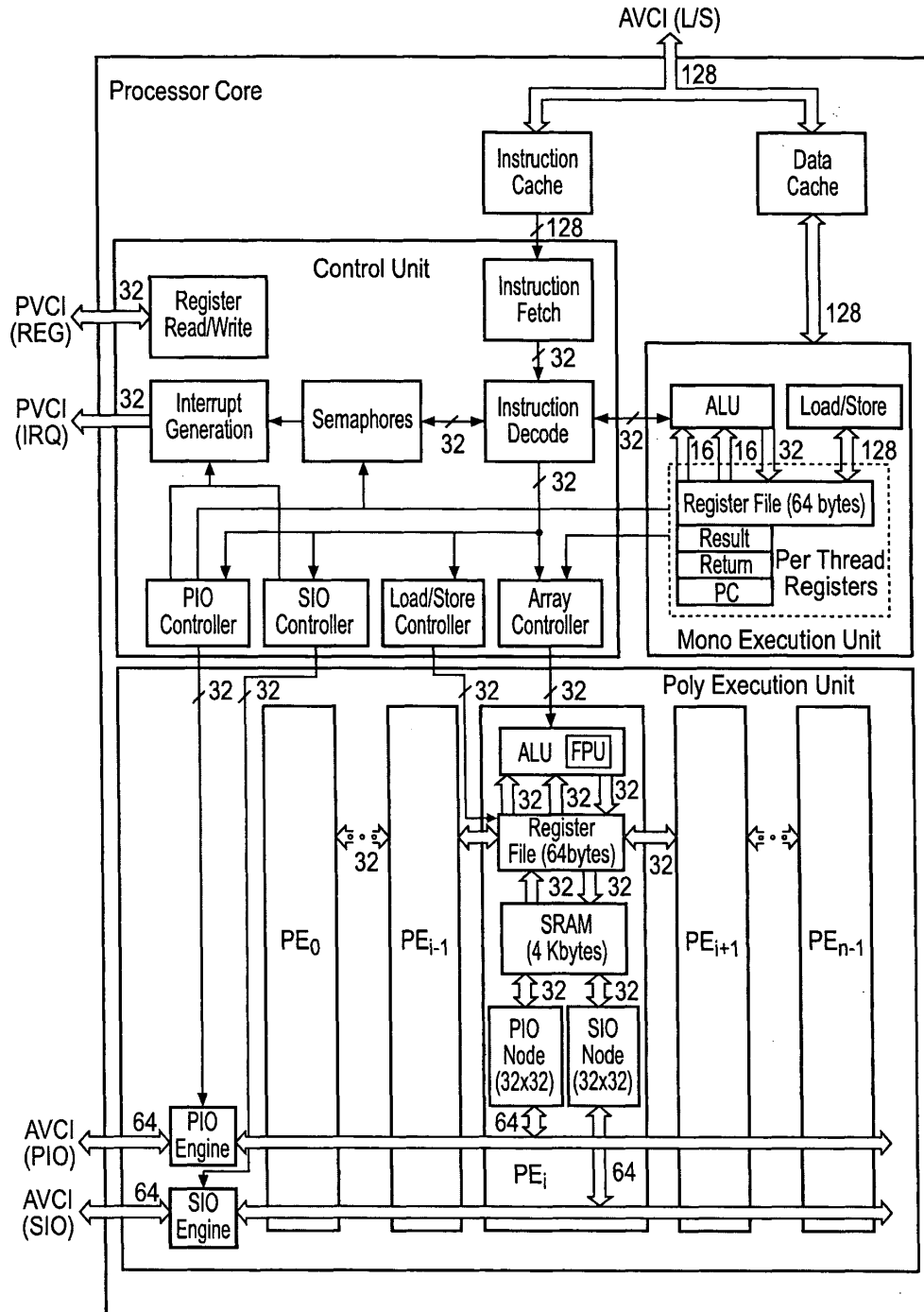


Fig. 7

6/6

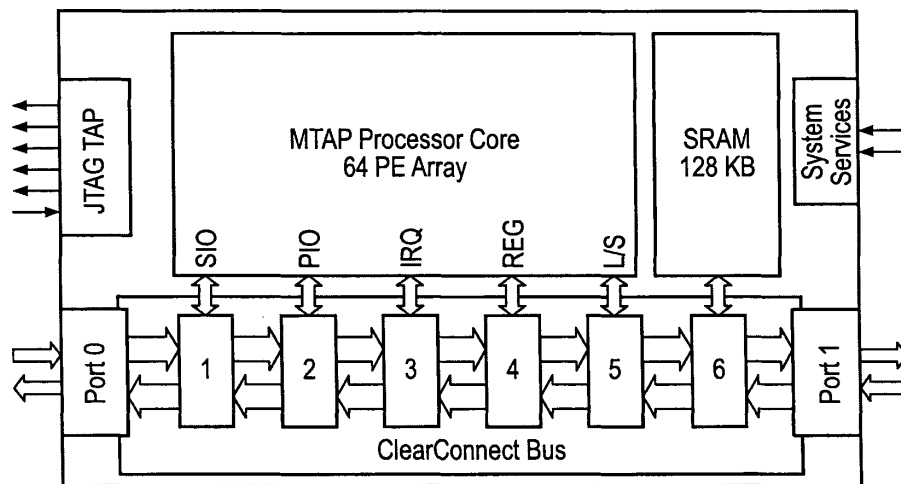


Fig. 8