Het debuggen van hard- en software in multi-processorstructuren

van den Broek, J.A.

Award date:
1980

TECHNISCHE HOGESCHOOL EINDHOVEN

AFDELING DER ELEKTROTECHNIEK

VAKGROEP DIGITALE SYSTEMEN

HET DEBUGGEN VAN HARD-EN SOFTWARE IN MULTI-PROCESSORSTRUCTUREN

afstudeerverslag van J.A.van den Broek

Het afstudeerwerk is verricht onder leiding van
Ir W. de Lange (Philips)
Ir M. Stevens (EB)

januari 1980

PHILIPS

Inhoudsopgave pag

MULTI PROCESSING

NAME NAAM Han v.d.Broek | SUPERS. VERV. | SH. BL. | SH. BL. —

PROPERTY OF EIGENDOM VAN N.V. PHILIPS' GLOEILAMPENFABRIEKEN·EINDHOVEN–NEDERLAND | CHECK CONTR. | DAT. | FORM. A4

INLEIDING

Voor u ligt het afstudeerverslag van J.A. van den Broek. Het afstudeerwerk is verricht in opdracht van de vakgroep "digitale systemen" van de Technische Hogeschool Eindhoven (vakgroep EB) bij de ontwerpgroep "PMDS" (Philips Microcomputer Development System), een onderdeel van de hoofdindustriegroep S&I (Science and Industrie) van de NV Philips, van maart 1979 tot februari 1980

In deze ontwerpgroep wordt een microcomputer ontwikkelsysteem ontwikkeld. U moet hierbij denken aan een minicomputer met een TV monitor, twee minifloppy diskdrives en een toetsenbord en met aansluitingen voor o.a. een line printer en een promprogrammer. Tot het software pakket horen behalve een diskette operating system o.a. eem texteditor, assemblers voor verschillende types microprocessoren, een link editor, een promprocessor en later ook een pascal compiler.

Een zeer belangrijk onderdeel van het PMDS is de zg "hardware debugger". Dit is een stuk logica dat, in samenwerking met de bijbehorende programmatuur, de ontwerpers van microprocessor systemen in staat stelt hun programmatuur en schakelingen verregaand te testen en van fouten te ontdoen. Vrij uniek is, dat in het PMDS vier hardware debuggers ondergebracht kunnen worden, zodat gelijktijdig maximaal vier processor systemen getest kunnen worden.

De huidige stand van de techniek maakt de microprocessors en de daarbij horende schakelingen steeds machtiger en goedkoper. Vandaar dat de tendens te verwachten valt om i.p.v. mini- en kleine computers, samenstellen van microcomputers te gaan gebruiken, zg multi- microcomputer systemen. Daarmee hangt dan samen het probleem om zulke systemen te ontwikkelen en te testen. Waarschijnlijk kan een MDS op dit gebied goede diensten bewijzen, mits hiervoor ingericht.

Het onderwerp voor mijn afstudeer opdracht is dan ook geboren uit het idee om door het installeren van meerdere debuggers in een MDS de verschillende microprocessoren in een multi-processor systeem gelijktijdig te kunnen testen en zodoende fouten in zo'n systeem te kunnen opsporen:

Allereerst moest er een onderzoek komen naar de verschillende manieren waarop een ontwerper een multi-processor systeem zou kunnen ontwerpen en naar de problemen die daarbij te verwachten zijn. Vanuit de inzichten die daaruit naar voren komen zou ik dan moeten komen tot een aantal aanbevelingen voor wijzigingen en uitbreidingen voor een toekomstige versie van het PMDS ten opzichte van het huidige, om zo adequaat mogelijk de ontwerpers van multi-processor systemen te kunnen ondersteunen.

Ik wil bij deze iedereen bedanken die me tijdens mijn afstudeerperiode geholpen heeft. Vooral alle collega's en medewerkers met wie ik de afgelopen tijd zo prettig heb samengewerkt en met name mijn beide coaches, de heren Stevens en de Lange van resp. de TH en van Philips, voor alle positieve bijdragen en steun die ze geleverd hebben en aan ir de Klerck die me in de gelegenheid heeft gesteld op zijn afdeling mijn onderzoek te verrichten.

PHILIPS

SAMENVATTING

Voor degenen die niet vertrouwd zijn met de begrippen "in circuit emulatie" en MDS is in dit verslag het derde hoofdstuk opgenomen. Dit beschrijft bovendien alle mogelijkheden van de hardware debugger zoals die op het ogenblik bestaat.

Bij het onderzoek naar multiprocessor systemen ben ik door het combineren van allerlei litteratuur (voor de belangrijkste wordt verwezen naar de litteratuur opgave) in staat geweest een overzicht op te stellen van de belangrijkste mogelijke systeem-configuraties. Dit overzicht is echt op systeem nivo. Als ik wil afdalen naar het hardware nivo om te kijken waar de problemen later zullen optreden bij het ontwerpen en testen stuit ik op een ontstellend gebrek aan concrete voorbeelden van bestaande multi- processor systemen. Deze systemen bestaan nog bijna niet, of zijn niet in de aanwezige litteratuur beschreven. Wel heb ik veel steun gehad aan het bestuderen van de iSBC 86/12, een single board computer die geschikt is voor gebruik in multiprocessor systemen.

Tijdens het bekijken van de mogelijkheden om multi-processor systemen te debuggen is me gebleken dat dit nu nog onvoldoende mogelijk is om te mogen stellen dat het MDS al geschikt is voor multi-ICE. De voornaamste reden daarvan is dat we nu niet in staat zijn om de software voor de meeste multi-processor systemen binnen het MDS te testen.

Het derde hoofdstuk is voornamelijk gewijd aan de beschrijving van de features die nodig zijn om deze software wel goed te kunnen testen en de wijzigingen van het PMDS die mogelijk zijn om deze te realiseren. Het testen van software zal waarschijnlijk de belangrijkste toepassing worden van het zogenoemde "multi-ICE".

Het laatste hoofdstuk bevat de beschrijving van enkele wijzigingen en toevoegingen aan de debuggers die het gemakkelijker a ken fouten op te sporen in bestaande gebruikers systemen. Deze wijzigingen zijn in het algemeen niet zo ingrijpend en ook niet allemaal per se noodzakelijk, maar wel handig.

Uit mijn onderzoek valt te concluderen dat een MDS een zeer belangrijk hulpmiddel zal worden bij het debuggen van multi- microprocessor systemen. Mits voorzien van de door mij gedane voorstellen is het er ook erg geschikt voor. Sommige extra voorzieningen zijn onontbeerlijk, maar vrij goedkoop aan te brengen. Van andere moet het praktisch nut worden afgewogen tegen de ermee gepaard gaande meerkosten. Bovendien zal, naarmate de complexiteit van het apparaat toeneemt, de fabrikant steeds meer genoodzaakt worden de gebruikers ervan uitgebreid te instrueren gebaseerd op een goede ontwikkelstrategie.

I MULTIPROCESSOR SYSTEMEN

In dit hoofdstuk zal het begrip multiprocessorsysteem en alles wat daarmee samenhangt worden behandeld, teneinde de lezer een indruk te geven van de problemen die ontstaan als we in een dergelijk systeem fouten willen opsporen of het willen testen en om toegang te verschaffen tot de onderhavige litteratuur.

We zullen eerst het terrein waarop we ons gaan begeven afbakenen aan de hand van de volgende

1.1 Definitie

In dit verband beschouwen we systemen waarin meerdere (micro-) processoren onderling samenwerken om een aantal taken te voltooien Ze moeten hierbij onderling kunnen communiceren en ze kunnen beschikken over gemeenschappelijke faciliteiten zoals geheugen en/of randapparatuur. Er moet een enkel operating systeem zijn.

Dat er meerdere processoren aanwezig moeten zijn is natuurlijk vanzelfsprekend. Het wezenlijke is dat ze kunnen communiceren, d.w.z. er moeten berichten, dus gegevens uitgewisseld kunnen worden tussen de processoren in het systeem. Op welke manieren dit kan gebeuren zal later in dit hoofdstuk duidelijk worden.

1.2 Voor- en nadelen t.o.v. computers met een enkele processor

De redenen dat steeds meer systemen waarin meerdere kleine processoren zitten worden toegepast i.p.v. grote computers met e een enkele processor zijn:

Dat multi-processor systemen efficienter kunnen zijn, doordat processoren kunnen worden gekozen aangepast aan een bepaalde de (deel)taak. Een computersysteem kan een aantal zeer diverse taken hebben zoals het maken van berekeningen, het besturen van allerlei processen of bv communicatie met de buitenwereld via allerlei randapparatuur. Voor elk van deze taken kan een processor geinstalleerd worden die daar het meest voor geschikt is.

Als het aantal taken van het computersysteem in de loop der tijden toeneemt kan men door het installeren van o.a. extra processoren eenvoudig zorgen dat steeds de gewenste capaciteit aanwezig is zonder dat erg hoge investeringen gedaan moeten worden.

Een dergelijk systeem kan betrouwbaarder zijn, omdat het economisch verantwoord is om wat extra capaciteit in te bouwen, zodat als er een processor defect raakt zijn taken kunnen worden overgenomen door een reserve processor.

Doordat de hardware steeds goedkoper wordt, komt de prijs-prestatie verhouding steeds gunstiger uit in het voordeel van de multi-processor systemen. De tendens is op het ogenblik nog zeer duidelijk dat de processoren in betrekkelijk korte tijd goedkoper, sneller en machtiger worden. Daar komt nog bij dat er momenteel voor micro-processor systemen gecompliceerde IC's beschikbaar zijn die taken kunnen vervullen waarvoor in grote systemen een aanzienlijke hoeveelheid logica nodig is.

Voor niets echter gaat de zon op want, zoals we later zullen zien, is de systeem software enorm gecompliceerd. Om nl de noodzakelijke onderlinge communicatie goed te laten verlopen is een ingewikkeld stuk software nodig dat alles in goede banen leidt. Omdat bovendien de systemen aangepast worden aan de behoeftes zijn er weinig identieke systemen. Het is dus nauwelijks mogelijk universele software te maken, hetgeen nog eens extra op de kosten drukt.

De uitbreidbaarheid is niet onbeperkt, want elke toegevoegde eenheid zal de benodigde overhead vergroten en de snelheid van het geheel doen verminderen doordat grotere wachttijden ontstaan bij gemeenschappelijke faciliteiten zoals bussen, geheugen en randapparatuur.

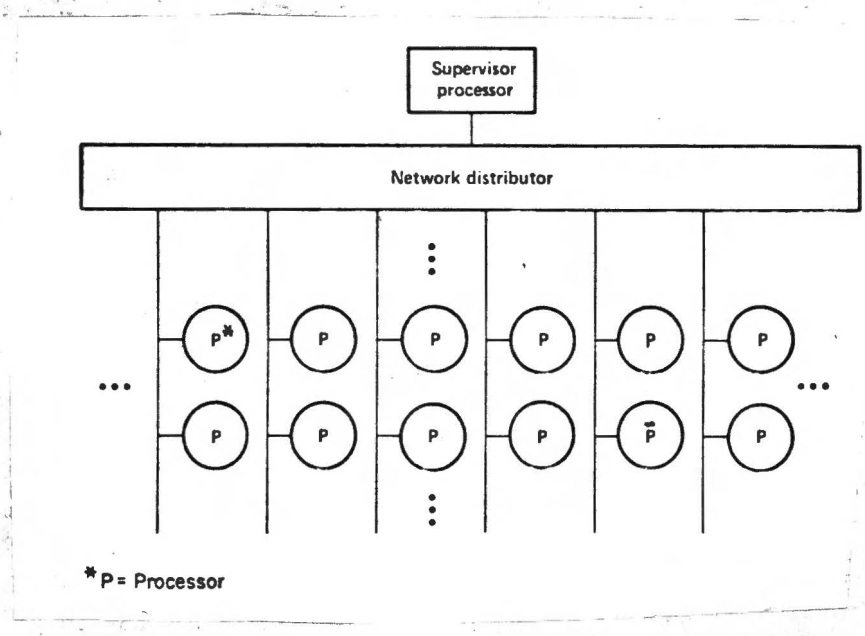
Het testen van de software wordt door de talloze mogelijke interacties erg tijdrovend. Het testen van software voor gewone systemen is al geen eenvoudige zaak. Bij multi-processor systemen moeten in principe alle mogelijke interacties tussen alle processoren getest worden voor men er zeker van kan zijn dat het totale pakket ook foutloos werkt.

1.3 Indeling van systemen

In de litteratuur treffen we vaak een indeling van computersystemen aan in vier categorieen. Deze hebben niet zo'n groot praktisch belang, maar de benamingen komen vrij vaak voor vandaar dat ik ze hier vermeld. Het zijn:

- Single Instruction Single Data (SISD): dit zijn de gewone computers met 1 processor. Ze nemen steeds 1 woord uit de totale gegevensstroom en voeren daar achtereenvolgens telkens 1 instructie gelijktijdig op uit.

- Single Instruction Multiple Data (SIMD): een veelgenoemd voorbeeld hiervan is de arrayprocessor. Bij een dergelijk type computer voeren een aantal parallel geschakelde microprocessoren steeds zelfde operatie uit op verschillende data, hetgeen vooral voordelen heeft bij vector- en matrix operaties. Een aparte processor zorgt voor het datatransport tussen de verschillende microprosessoren, en verzorgt de contacten met de buitenwereld.



Vanwege de zeer specifieke toepassingsmogelijkheden en de erg hoge snelheden van dit soort processoren zijn zij niet van belang voor onze beschouwingen omtrent de mogelijkheden tot emula-

- Multiple Instruction Single Data (MISD): processoren die gelijktijdig meerdere bewerkingen op dezelfde data uitvoeren lijken niet zinvol en bestaan dan ook niet. Dit type wordt slechts genoemd om de lijst compleet te maken.

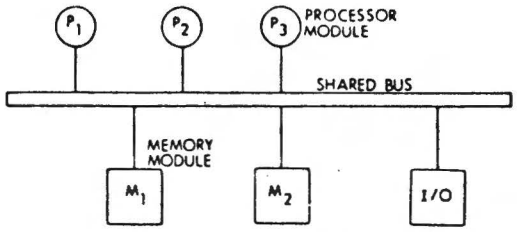
- Multiple Instuction Multiple Data (MIMD): met deze categorie computers, die dus gelijktijdig verschillende bewerkingen op verschillende data uitvoeren willen we ons in het vervolg bezig houden. We zullen nu nagaan op welke manieren ontwerpers dit soort systemen zouden kunnen opbouwen.

1.4 Hardware architectuur

Aan de hand van de interactie tussen de processoren kunnen we multiprocessor systemen onderverdelen in "tightly coupled"- en "loosely coupled" systemen. Alhoewel zo'n beetje alle artikelen een andere definitie van beide begrippen geven, wordt in grote lijnen gezien een systeem "tightly coupled" genoemd als de verschillende CPU's data uitwisselen via een gezamenlijk geheugen. D.w.z. dat er ergens in het systeem een stuk werkgeheugen moet zijn dat toegankelijk is voor meerdere processoren. De ene processor kan hier dan berichten in zetten, de ander kan ze lezen en daarmee is de communicatie dan een feit.

We kunnen een dergelijk systeem op drie principieel verschillende manieren in hardware opbouwen:

-Common bus structuur

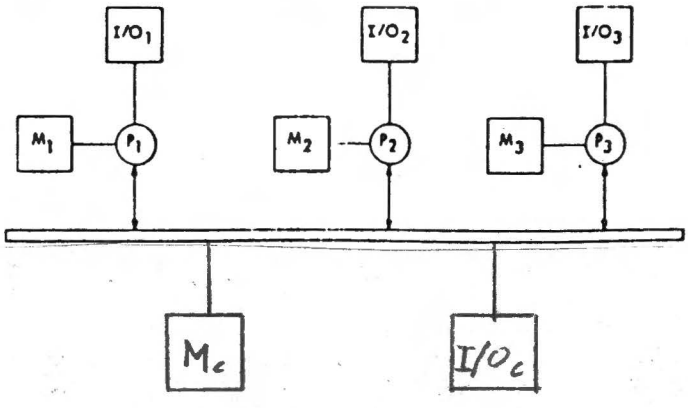


In het bovenstaande blokschemaatje ziet u de processormodules voorgesteld door cirkels. Hierin zijn ondergebracht de processor zelf met zijn buffers en drivers en met clock- en eventueel interrupt logica. Alle microprocessoren in het systeem zijn via buffers verbonden met een gemeenschappelijke, bijna altijd parallelle, bus. Deze omvat data-, adres-, handshake- en voedingslijnen. De hoeveelheid van elk is systeem gebonden en is helemaal afhankelijk van het ontwerp en de definitie van het totale systeem.

Eveneens aan deze bus zijn gekoppeld de geheugen modules en randapparatuur, hier voorgesteld door vierkantjes. In deze modules zit ook de benodigde selectie en decodeerlogica. Grootte en aantal van de eenheden is weer naar behoefte te kiezen.

De efficientie van een dergelijk systeem is vrij gering omdat de processoren slechts om de beurt kunnen werken. Er kan er nl altijd maar 1 tegelijkertijd van de bus gebruik maken.

Bij uitbreiding met extra processoren neemt de efficientie dus ook steeds minder toe naarmate er meer processoren aanwezig zijn. We zullen deze structuur dus alleen aantreffen bij kleine systemen. Een en ander kan verbeterd worden door een of meerdere processoren te voorzien van een eigen stuk geheugen en/of randapparatuur:



MULTI PROCESSING

NAME NAAM Han v.d.Broek | SUPERS. VERV. | SH. BL. | SH. BL. 1 — 4

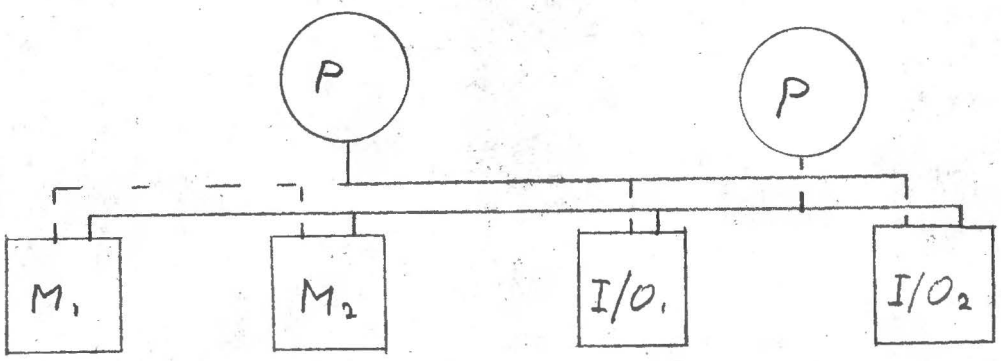
PROPERTY OF EIGENDOM VAN N.V. PHILIPS' GLOEILAMPENFABRIEKEN·EINDHOVEN–NEDERLAND | CHECK CONTR. | DAT. | FORM. A4

We zien dat er behalve de gemeenschappelijke geheugen en I/O modules ook per microprocessor een aantal van deze eenheden aanwezig kunnen zijn.

De processoren kunnen nu zelfstandig een aantal taken uitvoeren en hoeven slechts bij onderlinge communicatie en bij toegang tot achtergrondgeheugen en gezamenlijke randapparatuur van de bus gebruik te maken. De verwerkingssnelheid zal hierdoor hoger worden, maar nu wordt er inefficient gebruik gemaakt van randapparatuur. Immers een apparaat dat gekoppeld is aan processor 1 kan nu niet gebruikt worden door processor 2, ook al wordt het op dat moment niet gebruikt door processor 1. Bovendien als er nu een processor uitvalt kan zijn taak niet worden overgenomen door een ander, omdat allen de defekte toegang heeft tot de programmatuur en de data in zijn eigen geheugen.

Een andere manier om een systeem op te zetten is de volgende:

-Multiport/multibus



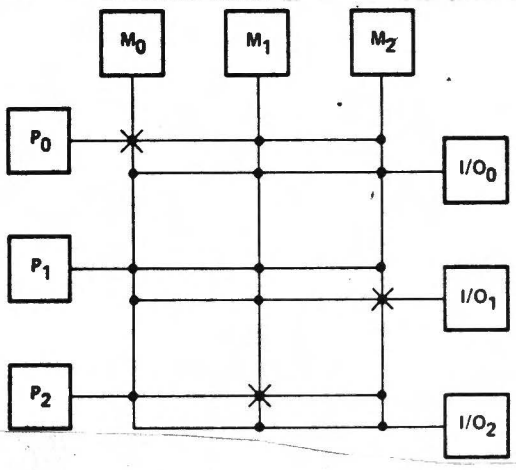
Bij deze configuratie is er voor elke processor een aparte bus. Deze bussen zullen in principe hetzelfde zijn als de bus uit het vorige geval. Op elk van deze bussen zijn een of meerdere randapparaten en stukken geheugen aangesloten via een aparte toegang (port). Deze ports dienen om te zorgen dat de bussen altijd gescheiden blijven en dat altijd slechts 1 processor gelijktijdig in een bepaalde module werkzaam kan zijn.

Niet elke processor hoeft nu toegang te hebben tot elk randapparaat of elk stuk geheugen, alleen de modules die aan zijn bus hangen zijn voor hem toegankelijk. Bovendien is het ook hier mogelijk dat processoren eigen geheugen en eigen randapparatuur hebben, hoewel dit nauwelijks interessant is, want we zien dat elke processor bijna op volle snelheid kan werken, tenzij hij net toegang wil tot een eenheid die al bezet is door een andere processor.

Werd bij de common bus structuur bij elke processor bepaald of hij op een bepaald moment al dan niet toegang tot de bus kreeg, hier vindt de arbitrage plaats bij de geheugen modules en bij de randapparatuur, waardoor deze omvangrijker en duurder worden dan in enkelvoudige systemen of in de eerder beschreven structuur. De uitbreidbaarheid wordt hier sterk beperkt door het aantal ports dat een geheugenmodule bezit (of kan bezitten). Ook is het duur aan bekabeling en connectors.

De derde van de genoemde "tightly coupled" systemen is de zg.

-"Crossbar switch" matrix



Via een uitgebreide set van schakelaars (elk kruispunt in het netwerk moet u zien als een bus-schakelaar) kan elke processor worden verbonden met elke willekeurige andere eenheid . U zult begrijpen dat dit systeem bizonder gecompliceerd is en dat het sturen van alle schakelaars enorm veel overhead vraagt. Daar staat tegenover dat alle andere eenheden eenvoudig en goedkoop zijn omdat er geen arbitrage en schakelfuncties door worden verricht. De uitbreidbaarheid wordt slechts beperkt door de technische grenzen van de schakelmatrix en bij elke uitbreiding wordt de efficientie van het systeem bijna evenredig vergroot. Ook bij uitval van een processor kan deze, tenminste wat de hardware betreft volledig worden vervangen door een ander.

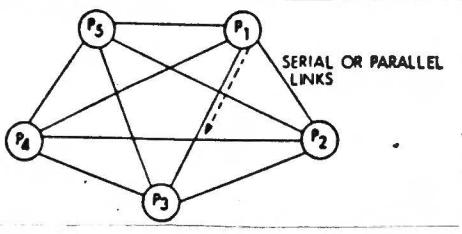
Natuurlijk hoeft ook hier niet elke processor met elke andere eenheid te kunnen worden verbonden. Er bestaan ook hele grote systemen die bestaan uit een aantal matrices als hier beschreven die onderling gekoppeld zijn via communicatie processoren in een hoger netwerk.

Ook bij "loosely coupled" systemen kunnen we een aantal verschillende systeem configuraties tegenkomen. Deze systemen zullen we aantreffen als de communicatie minder intensief hoeft te zijn. De onderlinge berichten uitwisselingen zullen veelal op seriele basis plaats vinden omdat deze methode goedkoper is en er gezien de intensiteit van de transfers geen hoge snelheden worden verwacht.

De communicatie gaat hier als volgt: De ene processor stuurt een bericht over naar een tweede. Vaak zal dit bericht daar eerst in een buffer of queue worden opgeslagen. Door dit buffer uit te lezen heeft de tweede processor toegang tot het bericht en kan het verwerken.

We onderscheiden hier de volgende configuraties:

-Volledig verbonden systemen.

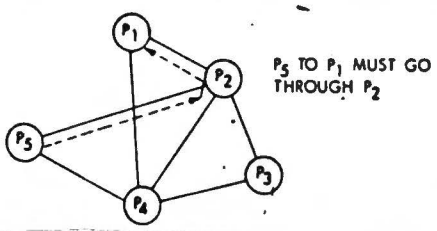


Merk op dat nu de cirkels complete computers voorstellen, dus processor, geheugen, eventueel randapparatuur en alle verder benodigde logica waaronder de eerder genoemde buffers.

Dit systeem wordt gekenmerkt door een direkte verbinding tussen alle computers in het systeem. Als de verbindingen serieel zijn, dan zal er vaak een aparte kabel zijn voor berichten de ene en de andere kant op.

Naarmate het aantal computers toeneemt zal dit systeem erg duur en complex worden. We kunnen dan kiezen voor een

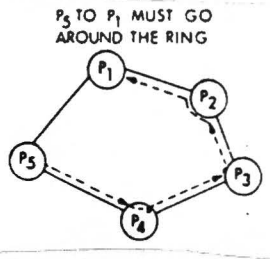
-Gedeeltelijk verbonden systeem



Dit is identiek aan het vorige met dit verschil dat nu niet alle mogelijke onderlinge verbindingen gemaakt zijn. Als nu bv C5 wil communiceren met C1, dan zal dit moeten gebeuren via C2 of C4. Elk bericht dat uitgestuurd wordt zal vergezeld moeten gaan van een code waaruit blijkt voor welke computer het bestemd is. Als een computer een bericht ontvangt dat niet voor hem is, zal hij moeten zorgen dat het in de goede richting doorgestuurd wordt. Hij wordt dus opgescheept met een extra stuk overhead dat zijn eigen werk zal vertragen. M.a.w. de software wordt duurder, de hardware goedkoper.

Als we dit principe van reduceren nog verder toepassen komen we tot:

-Ringstructuur

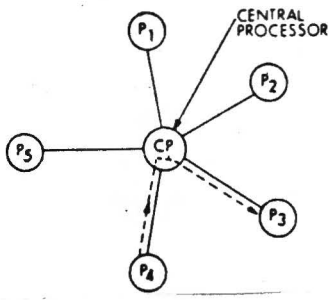


Hier kan communicatie tussen niet naburige computers alleen geschieden via tussenliggende computers. Een bericht wordt de ring ingestuurd en elke processor kijkt ernaar zodra het langskomt. Als het voor die processor bedoeld is wordt het uit de ring verwijderd, zoniet dan wordt het doorgestuurd.

De uitbreidbaarheid van een dergelijke configuratie wordt natuurlijk beperkt door de software overhead die er nodig is om alle berichten rond te sturen.

Tot nu toe waren alle processoren gelijkwaardig. Dit is niet het geval in de

-Sterstructuur

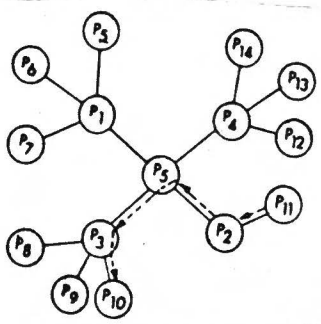


Hierbij is er een centrale computer aanwezig die onder andere, of uitsluitend bedoeld is om de communicatie te regelen tussen de andere computers. Alle berichten worden naar deze computer gestuurd, hij moet bepalen voor welke andere computer het bedoeld is en vervolgens zodra het kan doorsturen. Zodoende worden de zonderlijke computers verlost van de extra overhead, terwijl toch het aantal verbindingen minimaal is.

Het nadeel hiervan is dat er een extra computer nodig is voor de schakelfunctie en als deze uitvalt gaat het hele systeem plat.

Voor grotere systemen kan deze configuratie nog worden uitgebreid tot een

-Boomstructuur



Deze behoeft geen verdere toelichting.

U hebt kunnen zien dat er een groot aantal architecturen mogelijk zijn en door onderlinge combinatie van verschillende basis principes wordt dit aantal schier onbeperkt. Welk systeem het beste is hangt helemaal van de toepassing af en het is ook niet aan mij om daarover te oordelen.

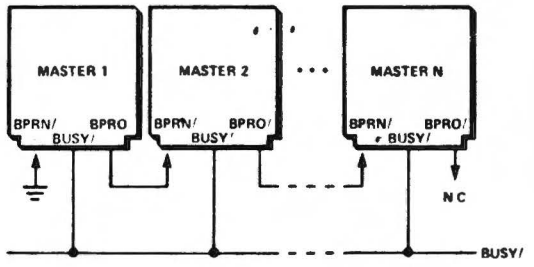
1.5 Arbitrage

Wanneer meerdere masters (dit zijn eenheden die toegang kunnen vragen en krijgen tot gezamenlijke faciliteiten, bv CPU's IO-processoren of DMA devices) gelijktijdig toegang verlangen tot een gemeenschappelijke faciliteit zoals bv de bus of multiport geheugen, dan dient er een schakeling, arbiter genaamd, aanwezig te zijn die zorgt dat er slechts een master gelijktijdig toegang krijgt tot de gewenste eenheid. Bij systemen met een gezamelijke bus moeten alle masters via buffers op de bus zijn aangesloten. De arbiter moet dan zorgen dat op elk moment slechts de buffers behorende bij 1 master opengestuurd zijn worden.

Bij slaves (eenheden die niet zelfstandig kunnen communiceren met de rest van het systeem, bv geheugens of randapparatuur, deze mogen overigens best intelligent zijn) met een multiport businterface is per slave een arbiter aanwezig die in dit geval maximaal een port openstuurt (een port is ook een set buffers).

Om bij meerdere aanvragen voor een bepaalde eenheid te weten welke master toegang moet worden verleend dient elke master een prioriteit te hebben zodat de arbiter diegene van de aanvragende masters met de hoogste prioriteit toegang kan geven. Het bepalen welke master de hoogste prioriteit heeft kan op meerdere manieren gebeuren bv:

-serieel ('daisy chain')



Per master is er arbitragelogica aanwezig. Hieruit komen in dit geval drie signalen naar buiten die de uiteindelijke toewijzing bepalen. Dit zijn:

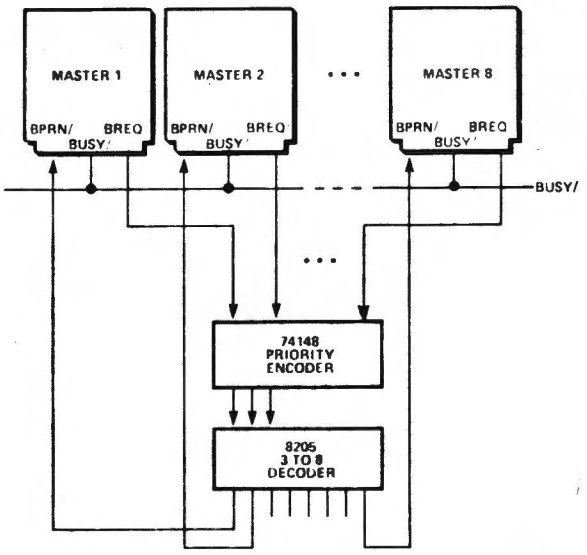
- priority in (PRIN). Als deze ingang laag is geeft dit aan dat er geen masters met hogere prioriteit een aanvraag doen.
- priority out (PROUT). Deze uitgang wordt hoog gemaakt als de betreffende master een aanvraag doet of als van dezelfde master PRIN hoog is.
- BUSY geeft aan of er wel of niet van de betreffende faciliteit gebruik wordt gemaakt.

Bij dit systeem bepaalt de plaats in de keten de prioriteit van de master: hoe meer naar links in de tekening, des te hoger is de prioriteit. Wanneer een master toegang wil tot de gemeenschappelijke faciliteit, zeg de bus, maakt hij zijn PROUT hoog. Dit heeft tot gevolg dat van alle masters met lagere prioriteit de ingang PRIN hoog wordt, zodat ze weten dat ze geen toegang hebben, ook al komt de bus vrij. Dit laatste is te zien aan de BUSY lijn. BUSY wordt geactiveerd door de master die op dat moment toegang heeft tot de bus. Zodra zijn datatransfer is beeindigd, geeft hij BUSY vrij. Dit wordt opgemerkt door alle andere masters. Degene die dan de hoogste prioriteit heeft activeert nu BUSY en krijgt de gewenste toegang.

MULTI PROCESSING

NAME NAAM Han v.d.Broek	SUPERS. VERV.	SH. BL.	SH. BL. 1 —10		
PROPERTY OF EIGENDOM VAN N.V. PHILIPS' GLOEILAMPENFABRIEKEN·EINDHOVEN-NEDERLAND			CHECK CONTR.	DAT.	FORM. A4

-Paralel



We vinden bij de hier aanwezige logica de signalen PRIN en BUSY terug, met dezelfde functie als in het vorige geval. REQ(uest) geeft door aan een priority encoder dat een betreffende master toegang wil tot de gezamenlijke bus. Nu bepaalt de plaats van aansluiting aan de priority encoder de prioriteit van de master. Deze encoder zorgt in samenwerking met een drie naar acht decoder (of een andere) dat maar een master het signaal PRIN ontvangt. Zodra de bus vrij komt grijpt deze master hem en neemt zijn REQ weg, zodat de volgende master PRIN kan ontvangen.

Het is, ook in het vorige geval trouwens, zinvol om de arbitrage synchroon aan een gezamenlijke klok te laten verlopen om problemen te voorkomen als twee masters gelijktijdig, of bijna gelijktijdig een aanvraag doen.

-Roterende prioriteit en andere

In het voorgaande zijn twee veel toegepaste arbitrage methoden behandeld, waarbij door de aansluitingen de prioriteit is vastgelegd. Dit laatste kan tot gevolg hebben dat masters met de laagste prioriteit(en) te weinig aan de beurt komen. Dan kan roterende prioriteits bepaling een uitkomst bieden. Hierbij krijgt een master die aan de beurt is geweest dynamisch de laagste prioriteit toegewezen. Dit kan gerealiseerd worden door ipv de priority encoder een sequentieel netwerk te gebruiken. Ook kan de prioriteit door de software bepaald worden. Aan de lezer wordt overgelaten om zelf nog andere vormen van prioriteitsbepaling te verzinnen.

-Speciaal geval

Bij kleine systemen (2 of 3 processoren kunnen we soms gebruik maken van de mogelijkheid om via een aansluiting op de chip (HOLD bij 8085, BUSRQ bij Z80) de processor te dwingen te stoppen met zijn taak en de bus tri-state te maken. Als dat gebeurd is komt er een terugmeldingssignaal (HLDA resp BUSAK) en weet de andere processor dat hij zonder gevaar op de bus kan. Deze lijnen zijn in de eerste plaats bedoeld voor DMA operaties, maar kunnen dus ook worden gebruikt voor een primitieve multiprocessor opzet. Ik zeg met opzet primitief, want als een processor op de gezamenlijke bus wil, moeten de andere stopgezet worden, onafhankelijk of zij wel of niet op de gezamenlijke bus willen. Tenzij er natuurlijk een schakeling is die dit detekteert en de HOLD / BUSRQ alleen doorgeeft als de tweede processor toegang vraagt tot de gezamenlijke bus.

1.6 Communicatie en synchronisatie

In het voorgaande is al herhaaldelijk gesproken over communicatie tussen processoren. Onder communicatie verstaan we in dit verband het uitwisselen van data tussen twee of meer processoren. Deze uitwisseling kan op meerdere manieren gebeuren nl

- via gemeenschappelijk geheugen

In een stuk geheugen dat toegankelijk is voor beide processoren zet de zendende microprocessor data klaar, die daarna gelezen moeten worden door de ontvangende processor. Deze data moeten op een tevoren afgesproken plaats komen te staan, of er moet op een bekende plaats worden aangegeven waar de data staan. De ontvangende processor moet ook weten wat hij ermee moet doen. Informatie hieromtrent kan worden meegestuurd met de data, of dit is bekend, bv omdat er maar een soort operatie kan worden uitgevoerd door de ontvanger, of het volgt uit de plaats in het geheugen waar de data wordt weggeschreven. Hetzelfde geldt voor de bewerkte data die teruggestuurd zullen worden, indien terugmelding vereist is.

-via I/O

Er kunnen een aantal buffers (fifo's) aanwezig zijn, waarin meestal een microprocessor kan schrijven en die een ander kan lezen, dus echt een verbinding tussen twee processoren. Dezelfde data die eerst via het geheugen werden doorgegeven gaan nu via buffers. Er is geen informatie meer nodig omtrent waar de data staan, wel zal, en zeker bij kleinere buffers, meestal al door de ontvangende microprocessor data gelezen moeten worden voordat de zender het hele bericht heeft weggeschreven, dit ivm de capaciteit van de buffers. De consequentie is dat de zender zal moeten kunnen testen of het buffer al dan niet vol is alvorens een (reeks) datawoord(en) te sturen.

Met synchronisatie bedoelen we hier software synchronisatie: De ontvangende microprocessor moet weten dat er data klaar staan die hij moet lezen en verwerken. Dit kan op een aantal verschillende manieren bekend worden nl:

-via interrupts

Zodra de zender een boodschap in het geheugen heeft klaargezet of een (aantal) datawoord(en) in het buffer, genereert hij bij de ontvanger een interrupt. Dit gebeurt in de regel doordat door de zendende processor een I/O poort wordt aangestuurd, waarvan de uitgang die direkt is verbonden met een interruptmechanisme van de ontvanger. Deze zal in veel gevallen een bericht terug moeten sturen om te bewerkstelligen dat de interrupt afgezet wordt. Wanneer een processor interrupts van verschillende bronnen kan onderscheiden, kan op die manier duidelijk zijn van welke zender een bericht komt, of waar in het geheugen een bericht te vinden is.

-via polling (ook "brievenbus" systeem genoemd)

In dit geval moet de zender op een bepaalde geheugenplaats (brievenbus genaamd) aangeven of er een bericht klaar staat. Dit gebeurt dan eenvoudigweg door op die plaats een bepaald bit of bitpatroon te zetten. De mogelijke ontvangers moeten dan op gekijken of er "post ligt". Het voordeel van dit systeem is dat er zette tijden kijken of er "post ligt". Het voordeel van dit systeem is dat er geen interruptbedrading hoeft te worden aangebracht. De software wordt echter gecompliceerder en de efficientie van het systeem neemt af, omdat lopende programma's steeds moeten worden onderbroken. De momenten waarop moet worden gekeken in de brievenbus worden overigens vaak bepaald door interrupts van een real time clock, zodat dit systeem dan toch weer veel lijkt op het vorige.

1.7 Kritische secties

Een bekend probleem bij het gezamenlijk gebruiken van geheugen of randapparatuur door meerdere processoren of door simultaan werkende programma's is dat het ene programma bepaalde data, waarmee een ander aan het werken is kan veranderen, zodat de ander niet meer reageert zoals verwacht. Een stuk geheugen of programma waarin deze fout kan optreden noemen we een kritische sectie.

Er zijn een aantal oplossingen bedacht om dit probleem het hoofd te bieden. Bv kan een kritische sectie worden beschermd door een semafoor (seinpaal). In zijn eenvoudigste gedaante is dit een geheugenplaats of I/O buffer, waarin d.m.v. een bitpatroon wordt aangegeven of er een programma in de kritische sectie bezig is. Voordat een programma de sectie binnen gaat moet het de semafoor testen om te zien of de sectie vrij is. Zoniet dan moet het blijven testen totdat de semafoor door een ander programma gereset is. Als de sectie vrij is dan moet het programma de semafoor setten en kan zijn gang gaan. Bij het verlaten van de kritische sectie moet de semafoor weer gereset worden. Gebeurt dit niet, bv omdat een programma een kritische sectie verlaat tgv een interrupt, dan kan "deadlock" optreden. Dwz. alle processen staan te wachten bij een semafoor en er wordt er geen gereset, dus ze blijven ten eeuwigen dagen wachten.

Wanneer we in de software de fout maken om niet te zien dat een bepaalde sectie kritisch is, of om niet de semafoor te testen die de sectie bewaakt, dan is er geen kruid gewassen tegen de fouten die hieruit kunnen resulteren.

Toch kan er nog iets fout gaan, ook als we netjes naar de semaforen kijken nl: Stel dat programma A de semafoor leest, gaat kijken of de sectie vrij is, blijkt dit het geval, dan zal hij de semafoor setten en de kritische sectie binnen gaan. Maar.... meteen nadat programma A de semafoor gelezen heeft, doet B dit ook. Het concludeert ook dat de sectie vrij is, omdat A nog geen tijd heeft gehad om de semafoor te setten, en gaat ook de kritische sectie binnen. Dijkstra heeft een software methode gevonden om ook dit probleem te omzeilen. Deze is echter noodzakelijkerwijs nogal omslachtig. Bij de nieuwere microprocessors zijn er tegenwoordig voorzieningen aanwezig die een gemakkelijker oplossing bieden nl. Voordat een processor de semafoor gaat testen kondigt hij een "buslock" af. Dwz. aan de arbiter laat hij weten dat hij tot nader order de bus wil en dat niemand anders er op mag. Dan kan hij de semafoor testen en zonodig setten, waarna de buslock weer opgeheven dient te worden. Dit heeft tot gevolg dat de efficientie van het systeem weliswaar iets afneemt, maar het testen en eventueel setten kan ongestoord gebeuren. Het is duidelijk dat onmiddelijk hierna de buslock moet worden opgeheven. Deze methode is alleen toepasbaar als alle processoren die in een kritische sectie kunnen werken op een aparte processor lopen.

Sommige processoren beschikken over een zg test en set instructie. Dit is een ondeelbare instructie die het bovengenoemde verzorgt.

In sommige gevallen kan een semafoor de gedaante hebben van een queue. Alle processen die in de sectie willen zetten hier een boodschap in, het operating systeem laat degene met de hoogste prioriteit toe zodra de sectie vrijkomt.

1.8 Operating sytems

In het begin van dit hoofdstuk zijn enkele voordelen van een multiprocessor systeem genoemd. Deze voordelen zijn natuurlijk alleen aanwezig als ze ook door de software worden ondersteund. Zoals reeds vermeld is deze software erg gecompliceerd en systeem gebonden. Toch wil ik hier volledigheidshalve enkele taken van het operating system behandelen nl:

-Het toewijzen van taken

Als je echt efficient gebruik wilt maken van een multiprocessorsysteem, moet je zorgen dat een bepaalde taak of deeltaak op die processor wordt uitgevoerd, die er het meest geschikt voor is of die op het gewenste moment vrij is. Het onderkennen van de geschiktste processor en het toewijzen van taken aan processoren is een taak van het operating system, en wel een erg moeilijke. Deze taak wordt vaak omzeild ten koste van een stuk flexibiliteit door elke gebruikerstaak door een vaste processor te laten verwerken.

-Het zoeken van paralellisme

Een taak kan soms erg snel en efficient worden uitgevoerd als hij wordt opgedeeld in een aantal subtaken die parallel kunnen worden verwerkt door verschillende processoren. Het maken van de meest geschikte opdeling van een taak moet gedaan worden door het operating system,of door een compiler. Hoewel een aantal mensen zich hier al uitputtend mee hebben beziggehouden is het nog niet gelukt om een goed algorithme te vinden of om een taal te definieren die een oplossing biedt voor dit probleem.

-Het detekteren en herstellen van fouten

Door redundantie te leggen in de boodschappen die door de processoren onderling worden uitgewisseld is het mogelijk te onderscheiden waar en wanneer er fouten optreden. Zoals we gezien hebben kan in een multiprocessorsyteem in sommige gevallen (afhankelijk van de architectuur) de invloed van deze fouten verminderd worden door de taak van het foutieve onderdeel te laten overnemen door een andere eenheid (zg gracefull degradation). Voorwaarde is dan dat het operating system kan detekteren welk onderdeel de fouten veroorzaakt, dat het kan bepalen door welk onderdeel de taak moet worden overgenomen, op welk punt hij moet worden hervat en kan zien dat de taak ook inderdaad op het vervangende onderdeel wordt afgewerkt.

-Het beheren en toewijzen van gemeenschappelijke faciliteiten

Als meerdere processoren gebruik willen maken van gemeenschappelijk geheugen, zal er een mechanisme moeten zijn dat aangeeft en bijhoudt welke processor welk stuk geheugen gebruikt en welke stukken er vrij zijn voor een volgende gebruiker. (memory managment) Hetzelfde geldt voor gemeenschappelijke randapparatuur.

II HET ONTWERPEN EN TESTEN VAN MICROPROCESSOR SCHAKELINGEN

2.1 Inleiding

Als u besluit in een electronische schakeling een of meerdere microprocessoren op te nemen kunt u niet meer volstaan met het slechts bouwen van de schakeling. U zult ook programmatuur voor die microprocessor(en) moeten schrijven en moeten testen. Deze programma's schrijft u dan natuurlijk het liefst in een vorm die u het meeste aanspreekt. Later zal er dan een "vertaling" moeten worden gemaakt van het door u geschrevene naar een door de microprocessor te verwerken code. Bovendien kunt u voor het testen van uw schakeling niet meer volstaan met de meetmethoden- en apparaten uit de conventionele electronica.

Op het ogenblik wordt voor de bovengenoemde taken vaak gebruik gemaakt van een zogenaamd "Microcomputer Development System", of kortweg MDS.

2.2 Wat is een MDS ?

Een MDS is zelf een computersysteem. Het bestaat in het algemeen uit een een toetsenbord via hetwelke de gebruiker teksten en commando's kan intypen; een beeldscherm, vergelijkbaar met dat van een draagbare TV. Hierop verschijnt al hetgeen de gebruiker typt en ook de informatie die van de computer verlangd wordt. Er is een achtergrondgeheugen in de vorm van vaste, of verwisselbare ('floppy disk' of diskette genoemde) magnetische schijven. Deze kunnen grote gebruikersprogramma's bevatten en ook de systeemprogramma's die door de fabrikant bij het MDS worden geleverd en die nodig zijn om het systeem zijn taken te laten vervullen. Programma's die op een diskette staan kunt u buiten het systeem bewaren en ten alle tijden in de computer inlezen om weer te gebruiken.

Alle gewenste informatie kunt u behalve op het beeldscherm zichtbaar laten maken ook op papier laten afdrukken door een zg. 'printer' die op elk MDS kan worden aangesloten. Soms zijn ook voorzieningen aanwezig om met ponsbanden te kunnen werken of met digitale cassetterecorders.

Het hart van een MDS wordt gevormd door een micro- of minicomputer. Deze heeft zelf een aantal (systeem)programma's nodig om te kunnen werken. De belangrijkste hiervan zijn:

Een operating system. Dit programma zorgt dat alle commando's die u intypt juist worden geinterpreteerd en, indien mogelijk, worden uitgevoerd. Ook beheert het het dikette systeem: programma's moeten op de juiste manier en plaats op een diskette gezet kunnen worden en ook vanaf de diskette in een geschikte werkruimte van het systeemgeheugen geladen kunnen worden. Verder zijn er nog een groot aantal operaties op diskette programma's mogelijk zoals vernietigen, veranderen, samenvoegen etc.

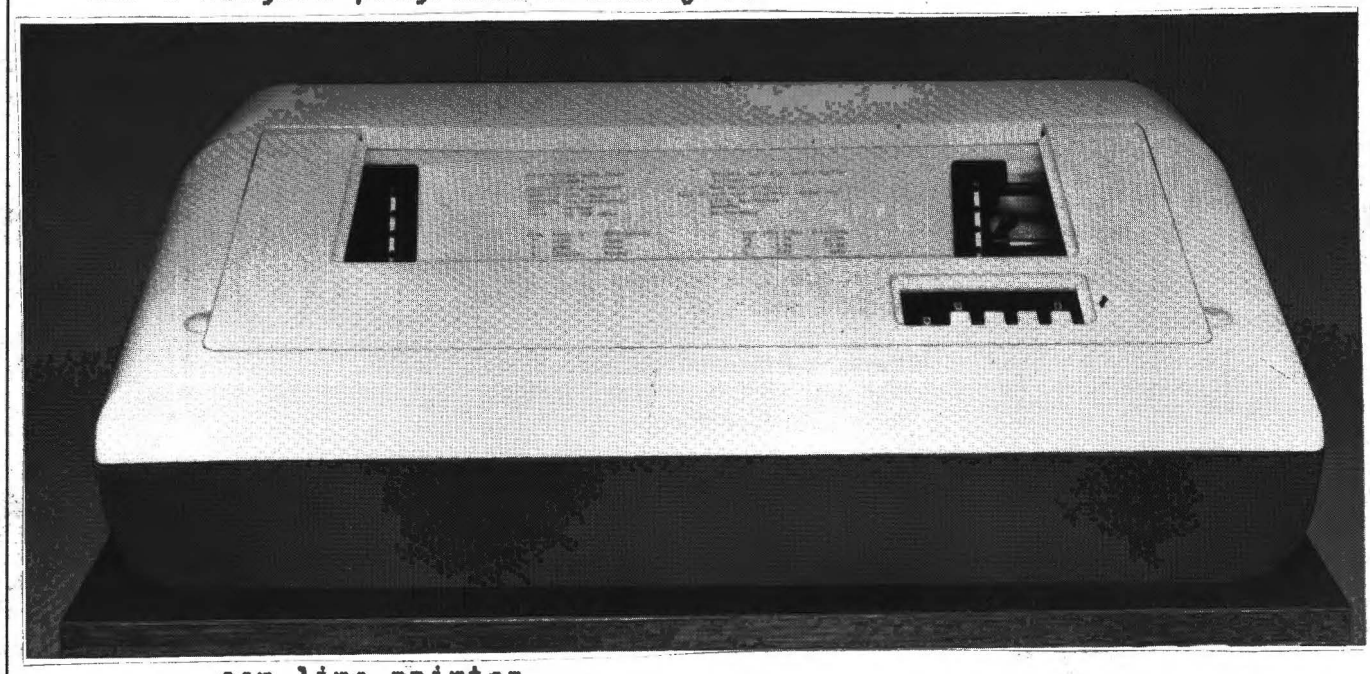
Een 'text editor'. Hiermee bent u in staat op eenvoudige wijze teksten zoals bv dit verslag in te voeren. U kunt naderhand heel gemakkelijk fouten verbeteren, regels tussenvoegen of verwijderen, bepaalde, eventueel eerder geschreven, stukken tekst tussenvoegen, bepaalde zinnen of zinsdelen opzoeken etc. en het geheel kunt u snel laten afdrukken op papier.

Deze teksten kunnen van alles zijn, maar zullen waarschijnlijk voornamelijk de 'source code' van programma's zijn. Dit wil zeggen op engels gebaseerde instructies, eventueel voorzien van commentaar. Deze programma's zijn dan bedoeld om straks uitgevoerd te worden door het computersysteem dat u zelf aan het ontwerpen bent. Hier volgt een voorbeeldje van zo'n source code die u zelf zou kunnen intypen, bedoeld om 1+2 uit te rekenen:

```
instructie          mogelijk commentaar

 MVI A,1        MVI='MoVe Immediate' dwz. zet 1 in de accumul.
 MVI B,2        zet 2 in register B
 ADD B          tel de inhoud van B bij de accumulator op
```

Deze instructies zijn nauwkeurig gedefinieerd voor elk type microprocessor, het commentaar is alleen makkelijk voor uzelf, als u langere programma's schrijft.



een line printer

Assemblers. Een assembler heeft tot taak de source code van een programma te vertalen naar een "object code". Dit is een vorm die geschikt is om te verwerken door een microprocessor. Eventueel foutieve instructies worden door de assembler als zodanig aangegeven, zodat u die met de text editor kunt verbeteren. Bovendien hoeft u niet te weten op welke plaatsen in het geheugen de instructies later komen te staan, u kunt deze adressen symbolische namen geven. Dit is zeer waardevol bij het werken met "jump"- en "call" instructies. Als het adres van de eerste instructie bekend is berekent de assembler de waardes van alle symbolische adressen.

Het eerder beschreven source programmaatje krijgt na assembleren de volgende vorm:

```
adres  obj. code   regelnr  instructie          commentaar

0000     3E01         1       MVI A,1
0002     0602         2       MVI B,2      commentaar onveranderd
0004     80           3       ADD B
```

Omdat elk type microprocessor zijn eigen instructies heeft, moet er ook voor elk type een eigen assembler aanwezig zijn.

Prom processor. Als aan uw MDS een zg. "prom programmer" is aangesloten kunt u met behulp van de prom processor de object code zoals die door de assembler is gegenereerd in een PROM IC laten zetten. Wanneer dit PROM later in uw schakeling is opgenomen kan uw microprocessor het beschreven programma uitvoeren.

een prom programmer

Er zijn meestal nog enkele systeem programma's, maar die zullen we hier verder buiten beschouwing laten.

Met hetgeen tot nu toe beschreven is kunt u met uw MDS een programma schrijven dat syntactisch foutloos is en dit via een PROM in uw schakeling brengen. Of uw programma ook logisch juist is en of uw schakeling zelf zonder fouten werkt is op dat moment nog maar de vraag.

Het MDS biedt u de mogelijkheid deze zaken grondig te testen. We gebruiken daarvoor een techniek die wel "in circuit emulation" wordt genoemd.

2.3 Wat is In Circuit Emulatie ?

In Circuit Emulatie, of kortweg ICE is een methode om de werking van een microprocessor in een schakeling te kunnen analyseren en controleren en van fouten te ontdoen. We noemen dit debuggen. Essentieel hierbij is dat de microprocessor uit de schakeling wordt verwijderd en dat de schakeling via de vrijgekomen socket wordt verbonden met een testapparaat, in ons geval een MDS.
De microprocessor wordt nu in het testapparaat gezet en kan daar zodanig werken dat het vanuit de schakeling gezien lijkt of de microprocessor nog steeds in de schakeling aanwezig is.

De emulatie houdt in dat we de microprocessor gewoon ("real time") of "single step" laten lopen. Via de pootjes van het IC zijn we in staat om alle benodigde informatie op te pikken. Deze kunnen we dan door het MDS in een zodanige vorm laten presenteren dat ze geschikt zijn voor analyse. Ook kunnen we bepaalde instructies gedwongen laten uitvoeren. Onder "real time" verstaan we in dit verband dat de microprocessor tijdens de emulatie dezelfde snelheid haalt die hij later in het definitieve ontwerp ook zal halen. Praktisch gezien betekent dit dat er geen extra "wait states" hoeven worden ingevoegd.

Het geheel van logica dat de ICE verzorgt wordt bij het PMDS "Hardware Debugger" genoemd. De programmatuur die zorgt dat de hardware debugger juist is ingesteld en de interactie verzorgt tussen operateur en de debugger heet de "debug processor".

Alvorens Multi-ICE te bespreken is het nuttig om te kijken wat de mogelijkheden zijn van de bestaande hardware debugger.

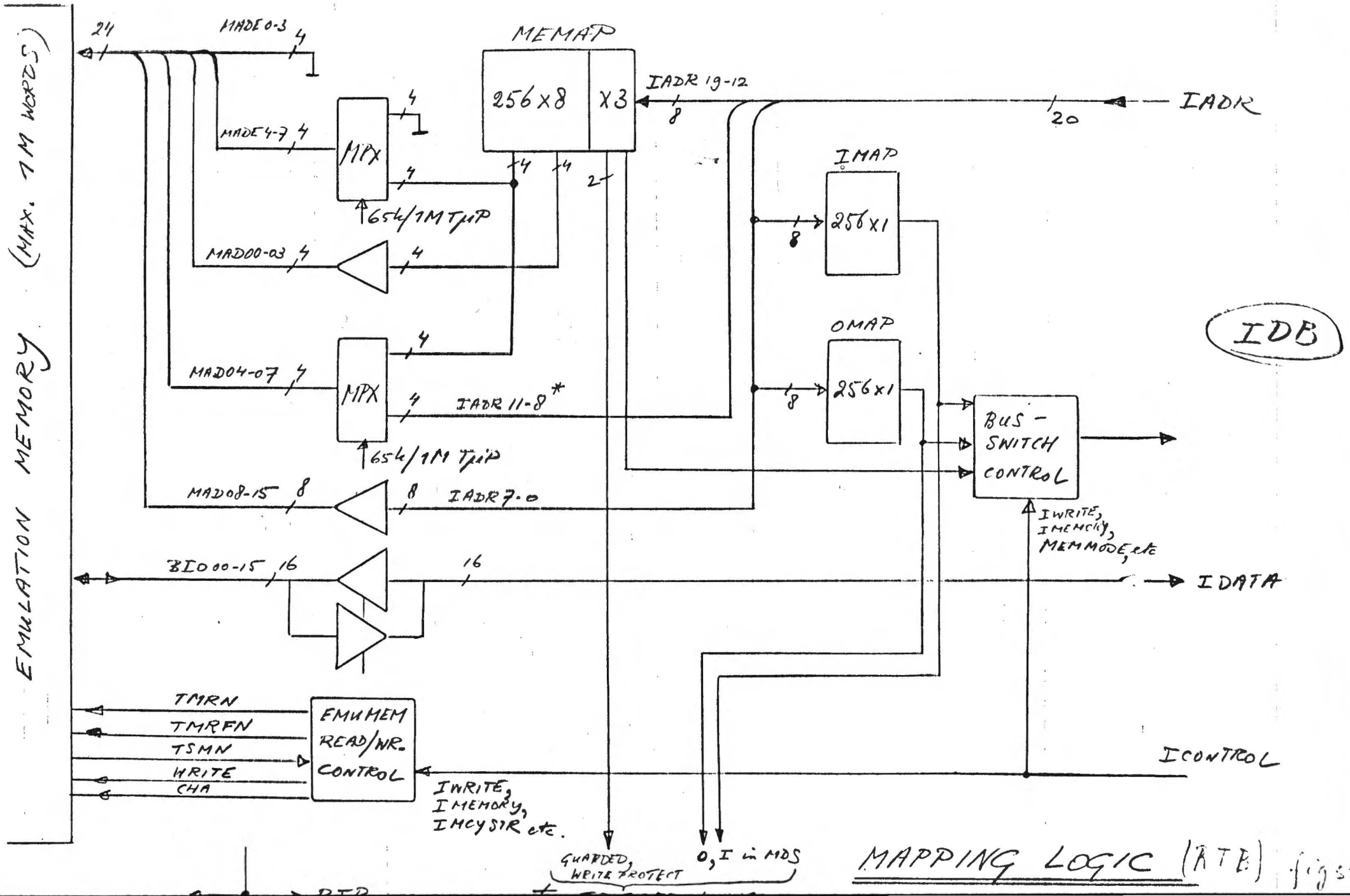
2.4 Speciaal geheugen in het MDS t.b.v. het debuggen.

Het kan voorkomen dat in onze schakeling op het moment dat we daar aan willen meten nog geen, of nog niet al het benodigde geheugen aanwezig is. In zo'n geval kan de te testen microprocessor, target microprocessor genaamd, gebruik maken van een speciaal stuk geheugen dat voor dit doel in het MDS aanwezig is, het zg. emulatiegeheugen. Dit is een stuk geheugen dat helemaal los staat van het werkgeheugen van de processor in het MDS. Het is alleen direkt toegankelijk voor de target processor en slechts indirekt voor de MDS-CPU (zie hiervoor par 2.8). We willen ook wel eens programma's testen zonder dat er al een werkende schaling klaar is. Dit kan zonder meer. We zetten dan onze programma's in hun geheel in het emulatiegeheugen.

De hardware debugger stelt ons in staat om de totale geheugenruimte van de target microprocessor in maximaal 256 evengrote blokken te verdelen. Bv bij de 8085 met een adresruimte van 64 k zal een blok dus steeds uit 256 opeenvolgende adressen bestaan. Van elk blok kunnen we nu apart opgeven of het al in het prototype aanwezig is of dat emulatiegeheugen moet worden gebruikt tijdens het debuggen. Dit laatste kan ook als in het prototype op de betreffende geheugenplaatsen fysiek geheugen aanwezig is. Het totale emulatiegeheugen kan 1 Megabytes omvatten.

We kunnen ook van elk blok opgeven door welk blok uit het emulatiegeheugen het moet worden geemuleerd. Een derde mogelijkheid is dat we een (aantal) geheugenblok(ken) "guarded" verklaren. Dan wordt de emulatie gestopt en verschijnt een foutmelding op het scherm zodra de target microprocessor probeert een adres uit de betreffende ruimte aan te spreken.

Ook kan van elk blok worden opgegeven of het ram of rom is. Pogingen om in een als rom gedefinieerd stuk geheugen te schrijven resulteren in stoppen van de emulatie en een foutmelding. Als de emulatie gestopt is kunnen we kijken op welke plaats in het programma de foutieve instructie staat en deze eventueel veranderen.


EMULATION MEMORY (MAX. 1M WORDS)
24
MADE 0-3
4
MEMAP
256 x 8
X3
IADR 19-12
8
IADR
20
MADE 4-7
4
MPX
4
654/1MTμP
4
4
4
2
IMAP
256 x 1
8
MAD00-03
4
4
OMAP
256 x 1
8
MAD04-07
4
MPX
4
IADR 11-8 *
654/1M TμP
BUS - SWITCH CONTROL
IWRITE, IMEMORY, MEMMODE, etc
MAD08-15
8
8
IADR 7-0
BIO 00-15
16
16
IDATA
TMRN
TMRFN
TSMN
WRITE
CHA
EMUMEM READ/WR. CONTROL
ICONTROL
IWRITE, IMEMORY, IMCYSIR etc.
GUARDED, WRITE PROTECT
0, I in MDS
IDB
MAPPING LOGIC (RTB)
figs

Doordat we voor het emulatiegeheugen zeer snelle rams kunnen gebruiken (zg "fast emulation memory") is de toegangstijd tot dit geheugen ondanks alle benodigde schakelhandelingen zo kort dat de targetmicroprocessor echt "real time" kan werken. Naar keuze kan ook het goedkopere, maar langzamere "slow emulation memory" worden gebruikt. Dit zal in het algemeen wel extra wait states vergen.

2.5 Simuleren van I/O operaties tijdens het debuggen.

Als tijdens het testen nog geen of niet alle randapparatuur aanwezig is kan het MDS de betreffende apparatuur simuleren. De totale I/O ruimte kan worden verdeeld in maximaal 256 input- en 256 output blokken. Bij bv een 8085 die maar 256 I/O poorten kent komt een blok overeen met 1 enkele I/O poort, zodat we voor elke poort afzonderlijk onderstaande gegevens kunnen specificeren.

Van elk blok kunnen we opgeven of het al dan niet in het prototype aanwezig is. Als dit het geval is dan zal de targetprocessor zijn I/O operaties op deze poorten tijdens de emulatie gewoon real time uitvoeren. Zo niet, dan kunnen we in geval input verlangd wordt, deze invoeren via het toetsenbord. Output kan in hexadecimale vorm worden zichtbaar gemaakt op het scherm. Een tweede mogelijkheid is dat de gewenste data onder een tevoren opgegeven filenaam van diskette worden gelezen resp naar diskette worden geschreven. Als geen van beide mogelijkheden worden gespecificeerd is het blok guarded, met dezelfde consequenties als bij guarded geheugen.

Zodra het programma van de targetmicroprocessor tijdens de emulatie een I/O poort aanspreekt die tevoren is opgegeven als zijnde "in MDS" stopt de emulatie. Als input wordt verlangd vanaf het toetsenbord verschijnt op het scherm een mededeling dat dit het geval is, met daarbij het adres van de betreffende poort. De gebruiker moet nu een hexadecimale inputwaarde intypen. De debugprocessor geeft deze door aan de targetmicroprocessor en daarna wordt de emulatie weer automatisch doorgestart. In geval van output via het scherm haalt de debugprocessor de data van de targetmicroprocessor en schrijft deze, samen met het adres van de outputpoort naar het scherm, waarna de emulatie wordt doorgestart. Ongeveer analoog wordt gehandeld bij diskette operaties.

De targetprocessor werkt dus in zo'n geval precies als hij in de uiteindelijke situatie ook zal doen, alleen wordt nu door de tussenvoeging van een (groot) aantal wait states de afhandeling van een programma vertraagd. Daar het ons in zulke gevallen gaat om de correcte werking van het programma en niet om het tijdsaspekt zal dit geen bezwaar zijn.

Als een guarded poort wordt aangesproken verschijnt een foutmelding op het scherm en blijft de emulatie gestopt.

PHILIPS

2.6 De logic state analyzer functie.

Een veel gebruikt maar vrij nieuw meetapparaat om bv microprocessor programma's te bekijken tijdens de uitvoering ervan is de zg Logic State Analyzer (LSA). U kunt dit het beste vergelijken met een veel kanaals (16-48) geheugen oscilloscoop. Op bepaalde momenten worden de signalen op de ingangen van de LSA gesampled. Als ze boven een bepaalde "drempelspanning" liggen worden ze als logische 1 opgelagen in het digitale geheugen van de LSA, anders als logische 0. Het moment van samplen wordt bepaald door een signaal uit de schakeling (veel gebruikt zijn de read- of de write strobe) dat wordt aangesloten op de zg "clock" ingang van de analyzer.

De waarden van de ingangen worden op elk clock moment ingenomen. Hiermee wordt gestopt als aan een "triggerconditie" is voldaan. Een eenvoudige triggerconditie zou kunnen zijn dat een bepaald "triggerwoord" is opgetreden, d.w.z. een tevoren opgegeven specifieke combinatie van nullen en enen op de ingangen van de LSA op een clock tijdstip.

Het kan zijn dat niet alle data die zo wordt ingenomen voor ons interessant is. We hebben dan de mogelijkheid om alleen data in te nemen als op de clock momenten aan een "qualifier" functie is voldaan. Dit betekent dat op zo'n moment bepaalde ingangen hoog, of juist laag moeten zijn, al naar gelang men dit tevoren opgeeft.

De presentatie op het scherm van de LSA zal vaak de waarden van de ingangskanalen in binaire vorm weergeven, of hexadecimaal door steeds de kanalen in groepen van vier te combineren.

Voor verdere informatie wordt verwezen naar mijn stage verslag over dit onderwerp. Dan volgt nu een beschrijving op welke manier het PMDS de logic analyzer functie vervult.

Als optie wordt bij de hardware debugger een kaart, het zogenaamde Real Time Trace Memory (RTTM) geleverd. Met deze optie kan het MDS fungeren als volwaardige logic state analyzer. De geheugendiepte is 256 woorden van 48 bits. Er zijn dus 48 kanalen die, als we aan een bepaald type microprocessor meten, allemaal een vaste functie hebben. De functie van deze 48 bits is afhankelijk van het type targetmicroprocessor:

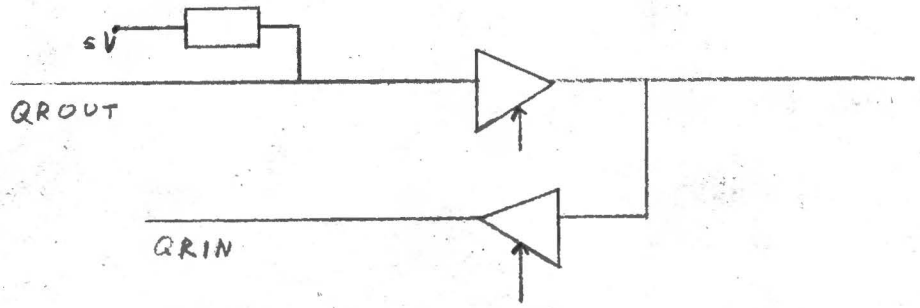
- alle adreslijnen worden bekeken, dit zijn er 12 bij de 8048,
  16 bij 8085 ea.
  20 bij 8086.
- 4 controle lijnen, bv bij de 8085: ALE, WR, RD en IO/M
- een aantal extra lijnen. Dit aantal is zodanig dat het samen met de adreslijnen 20 bits omvat. Bij de 8085 zijn dit er nog 4. Ze worden gebruikt voor HLDA, SOD, SID en SO.
- De datalijnen (8 of 16)
- Externe probes. Bij 16 bits microprocessors kunnen we 1 probe aansluiten, bij 8 bits microprocessors zijn dit er twee. Iedere probe bestaat uit 8 kanalen waarop willekeurige signalen van het prototype kunnen worden aangesloten. De drempelspanning is per probe software instelbaar in stappen van 50 mV tot een maximum van 10.5 V.

Als analyzer clock kunnen we 1 van de userprobe lijnen gebruiken, waarbij dan naar keuze op de opgaande of de neergaande flank gegevens worden ingenomen.

Andere mogelijkheden zijn dat er elke machinecycle of elke instructiecycle of elke opcode fetch cycle gegevens worden ingenomen. In deze gevallen wordt de informatie op de userprobelijnen ingeklokt op de achterflank van de laatste clockpulse van de betreffende cycle.

Data inname kan onvoorwaardelijk geschieden, maar kan ook geconditioneerd worden door twee qualifiers. Er zijn twee 48 bits qualifiers, die beide dezelfde indeling hebben als de ingenomen woorden. Van elk bit afzonderlijk kan worden opgegeven of het in de qualifiers 0, 1 of don't care moet zijn. We hebben de mogelijkheid om hetzij de ene qualifier te gebruiken, hetzij de andere, dan wel gegevens in te nemen als aan een van beide is voldaan (or-functie).

Bovendien is er nog een externe qualifier, die al dan niet meespeelt. Via de BNC connector waarop deze moet worden aangesloten kan ook naar buiten worden doorgegeven wanneer het RTTM gegevens inneemt. Door nu de externe qualifiers van de RTTM's van de verschillende hardware debuggers onderling te verbinden bereiken we dat ze alle gelijktijdig gegevens innemen. Deze qualifier is als volgt geschakeld:



Normaal is de gemeenschappelijke lijn hoog. Als een van de RTTM's gegevens inneemt maakt hij zijn QROUT lijn laag. Als deze enabled is wordt de gemeenschappelijke lijn ook laag. Alle RTTM's waarvan QRIN enabled is nemen dan op dit moment ook gegevens in.

Voor de triggering zijn twee triggerwoorden aanwezig. Wederom ieder 48 bits breed, met dezelfde indeling als de qualifiers en ook met de mogelijkheid om ieder bit 0, 1 of don't care te maken. Triggering geschiedt als het ene triggerwoord is gevonden, of als het andere is gevonden, of als het ene is gevonden nadat N keer het andere is opgetreden. N wordt hier bepaald door de inhoud van een 8 bits "match counter" waarvan we de inhoud tevoren moeten opgeven. Er zijn twee matchcounters zodat ook triggerwoord 2 gevonden mag worden nadat eerst M keer triggerwoord 1 is voorgekomen. Welke triggermogelijkheid, of combinatie van mogelijkheden wordt gebruikt is ook via het toetsenbord te kiezen. Als de emulatie stopt, stopt automatisch de tracing.

Tenslotte moeten we nog opgeven de wijze van presentatie nl. pre- centre- of post triggering. Disassembly wordt automatisch gepleegd.

2.7 Een teller voor het bijhouden van allerlei gebeurtenissen.

Op dezelfde kaart als het RTTM is een teller, de zg 'event counter' aanwezig. Dit is een 3 bytes' teller die we op of neer kunnen laten tellen of af kunnen schakelen. Deze kan gebruikt worden om tijdens de emulatie bepaalde gebeurtenissen te tellen. Daartoe kunnen we de tellerstand uitlezen als de emulatie gestopt is. We kunnen ook de teller laden met een bepaalde waarde en dan aftellen. Zodra de teller leeggeteld is stopt de emulatie.

De eventcounter kunnen we laten beginnen te tellen op:

- het begin van de emulatie
- triggerwoord 0
- triggerwoord 1
- triggerwoord 0 of 1

en laten stoppen op:

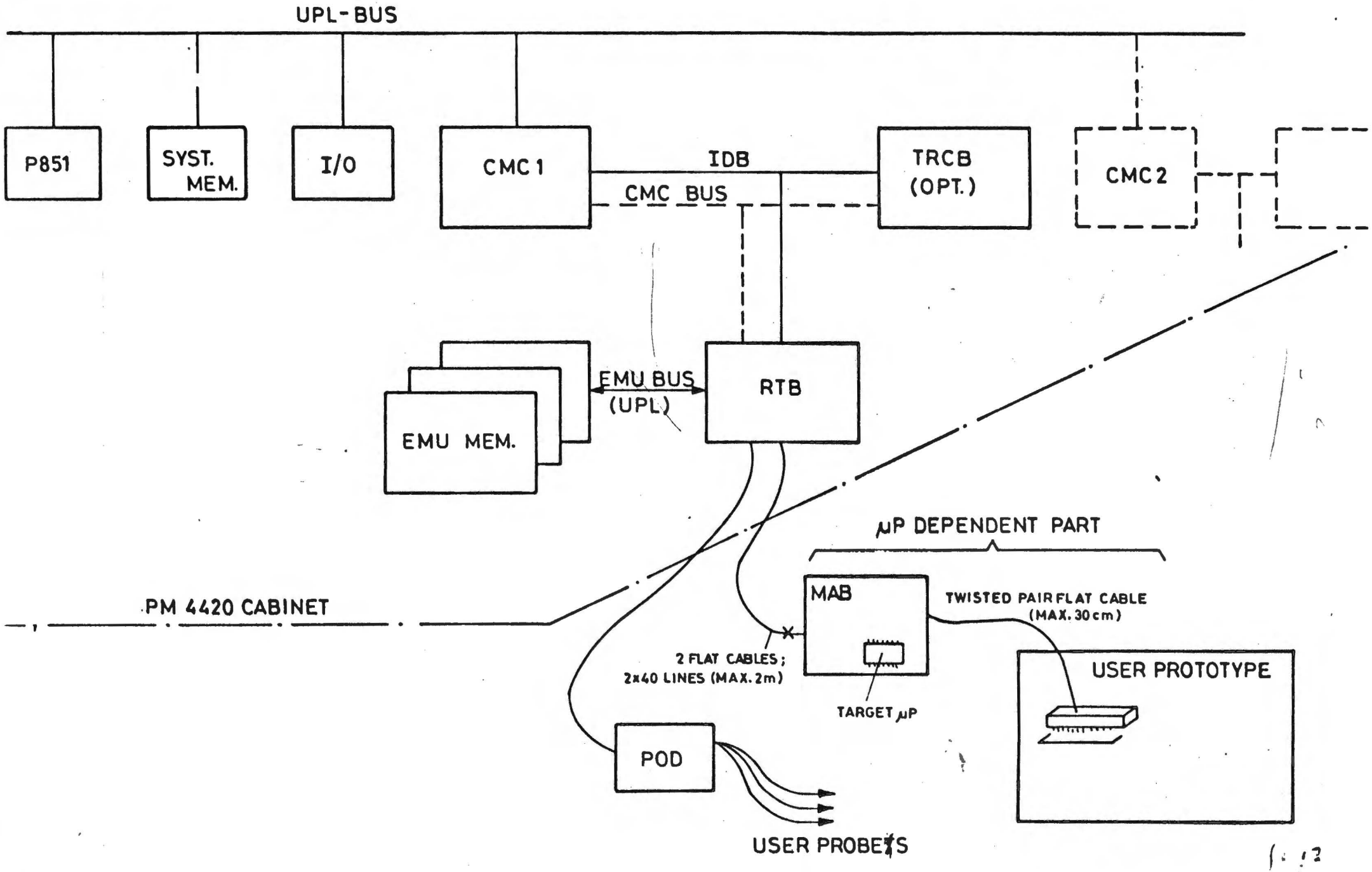
- het einde van de emulatie
- triggerwoord 0
- triggerwoord 1

Wat er geteld moet worden wordt ook via het toetsenbord opgegeven. We hebben de keuze uit:

- real time clock pulsen (1 micro seconde klok)
- clock pulsen van de target microprocessor clock
- machine cycles
- opcode fetches
- interrupt acknowledge cycles
- aantal keer dat triggerwoord 0 voorkomt
- aantal keer dat triggerwoord 1 voorkomt
- een signaal waarop een van de userprobes is aangesloten

Door deze teller op een gunstig gekozen signaal aan te sluiten kunnen we een indruk krijgen hoevaak een bepaalde gebeurtenis bv. een interrupt of een DMA-cycle of iets dergelijks tijdens een emulatierun optreedt.

UMDS STRUCTURE



2.8 Het laden van gebruikersprogramma's in emulatiegeheugen.

De programma's voor de targetmicroprocessor zullen we in het algemeen met behulp van het MDS schrijven in assembleertaal of in een hogere programmeertaal. De machinecode die we krijgen na assembleren resp compileren zetten we in een (p)rom, dat dan in het prototype komt. Voordat we dit (p)rom (laten) maken willen we natuurlijk wel eerst ons programma getest hebben. Dit kunnen we doen door het programma van de diskette te halen en in het emulatiegeheugen te zetten (downloading). Door nu de memorymapping zo in te stellen dat de targetprocessor dit programma ziet op de locaties waar later het (p)rom komt te zitten en door dit stuk geheugen te definieren als rom is het voor de targetprocessor precies of het betreffende (p)rom in de schakeling zit en kunnen we de gewenste tests uitvoeren. Eenvoudige veranderingen kunnen we zodoende snel aanbrengen, zodat het effect ervan meteen weer getest kan worden.

2.9 De eigenlijke emulatie

Als we alle voornoemde zaken hebben ingesteld voor zover ze voor ons van belang zijn voor een bepaald onderzoek zijn we bijna klaar om te gaan emuleren. We kunnen nog opgeven waar de targetprocessor zijn clock vandaan moet halen. De keuzemogelijkheden zijn:

- de clock in het prototype
- een extern aangebracht kristal
- een externe TTL klok
- een door het MDS gegenereerd signaal van ongeveer 1, 3, 6 of 9 Mhz

Voordat we de targetprocessor laten lopen kunnen we al zijn registers laden met een bepaalde waarde. Door de programcounter met een bepaalde waarde te laden kunnen we bepalen op welk adres de microprocessor zal beginnen te lopen. Met het commando "RUN" op het toetsenbord van het MDS kunnen we de targetprocessor zijn programma uitvoer laten starten (dus de emulatie starten). Deze zal stoppen of niet starten als aan een van de volgende voorwaarden is voldaan:

1) Er is aan een van te voren ingestelde stopconditie voldaan nl. van de 2 triggerwoorden en de qualifiers kan elk worden opgegeven of ze moeten worden gebruikt om de emulatie te stoppen zodra ze gedetekteerd worden.

Als een programma ergens fout gaat, en u wilt uitzoeken waar, dan kunt u op een bepaalde plaats in het programma een "breakpoint" zetten. Zodra de processor hier komt zal de emulatie stoppen. U kunt dan kijken of de processor via de juiste weg op die plaats is gekomen, of een bepaalde loop het juiste aantal keren is doorlopen, of alle registers de goede inhoud hebben etc. Fouten kunt u verbeteren en als alles tot nu toe goed is kunt u de processor verder laten lopen tot een volgend breakpoint.

2) Via het toetsenbord is het commando 'HALT' gegeven.
Dit zal voornamelijk gebeuren als u bepaalde breakpoints hebt opgegeven en de emulatie stopt niet, omdat de processor door een fout in uw programma niet op zo'n breakpoint komt.

3) Er is een foutconditie opgetreden.

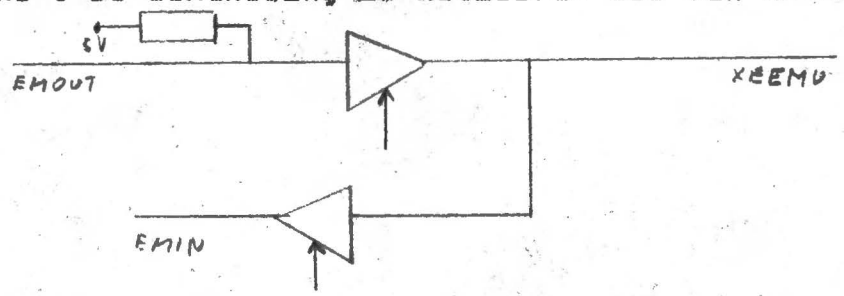
1 Foutcondities zijn in dit geval:

- Een time out van de eventcounter
- een poging om toegang te krijgen tot 'guarded' geheugen
- een poging om te schrijven in als rom gedefinieerd geheugen
- het prototype is niet aangesloten
- er wordt toegang gevraagd tot een I/O poort die 'in MDS' is verklaard
- de 'inter run control line' is actief (zie hieronder)
- van elk van de volgende condities van de targetprocessor kan worden opgegeven of ze al dan niet de emulatie moeten doen stoppen:
  * reset
  * not ready time out
  * DMA cycle
  * halt
  * clock fout
  * fout in de voedingsspanning

Als de emulatie gestopt is verschijnt op het scherm de reden dat hij gestopt is. We kunnen dan de registers uitlezen en eventueel de inhoud van het RTTM en de eventcounter. Ook kunnen we een willekeurige (reeks) geheugenplaats(en) van de target-processor opvragen. Met deze middelen moeten we in staat zijn om de oorzaak van een eventuele fout te vinden.

Zonodig kan een programma ook 'single step' worden uitgevoerd. Uiteraard gaat dit niet meer 'real time', maar we krijgen na elke instructie de inhouden van alle registers op het scherm.

Als er meerdere hardware debuggers zijn aangesloten bestaat de mogelijkheid om ze allemaal tegelijkertijd te starten of gezamelijk te stoppen. Dit geschiedt dmv. de zg 'inter run control line'. De schakeling is hetzelfde als van de externe qualifier:



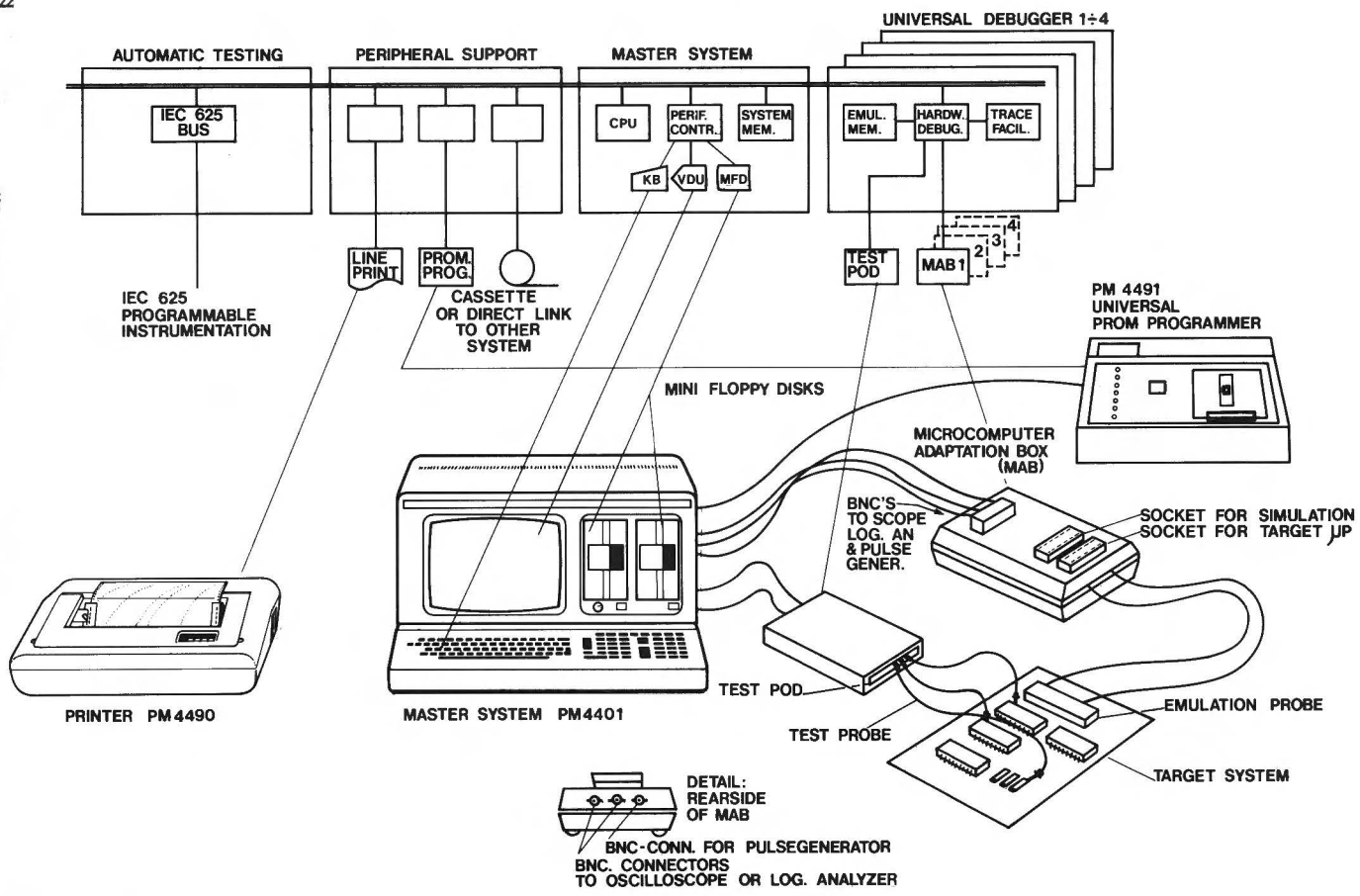
Als een emulatie gestopt is maakt de betreffende hardware debugger EMOUT laag. Als EMOUT enabled is wordt hierdoor XEEMU laag. Alle hardware debuggers waarvan EMIN enabled is stoppen nu ook. Een hardware debugger die klaar is om te starten maakt zijn lijn EMOUT floating. Als alle HWD's die we met elkaar willen synchroniseren (EMOUT enabled) gereed zijn om te starten maakt een pull up weerstand XEEMU hoog, waardoor alle HWD's waarvan EMIN enabled is zullen starten.

2.10 De opbouw van de hardware debugger

In verband met komende beschouwingen over eventuele aanpassingen zal ik in het kort iets vertellen over de opbouw van het huidige systeem en in het bizonder over de hardware debugger.

Er is een grote, nauwkeurig gedefinieerde, systeem bus, UPL (Unified Product Line) genaamd, hieraan zijn de verschillende eenheden aangesloten tw.

- De CPU, een P851 minicomputer met de bijbehorende logica.
- Systeemgeheugens 32k x 16 dynamische rams
- De keyboard interface met daaraan het toetsenbord zelf
- De VDU (video display unit) interface met daaraan het display
- Seriele interfaces voor line printer, prom programmer, verbinding met een INTELLEC systeem
- Opties zoals IEC-bus controller
- 1 Hardware debugger



Indien meerdere debuggers moeten worden gebruikt, zullen deze in een aparte kast worden ondergebracht. Op dit moment is het nog een afwegen van vooral kosten tegen snelheid of deze debuggers zullen worden aangestuurd door uitbreiding van de UPL bus of op seriele basis, waarschijnlijk zal de eerste mogelijkheid gekozen worden.

Als u met een emulatie begint zult u eerst de betreffende programmatuur (debug processor) in het werkgeheugen van het MDS moeten laden. Alle commando's die u dan verder intypt zullen door de debugprocessor geinterpreteerd en verwerkt worden en zullen uiteindelijk moeten resulteren in een correcte instelling en aansturing van de hardware debugger. De interface tussen de P851, waarop de debugprocessor draait, en de hardware die de emulatie verzorgt wordt gevormd door het "CMC board" (Control Micro Computer).

Via een programmed channel kan de P851 over de UPL commando's naar een buffer op het CMC board sturen. Op deze kaart bevindt zich een microprocessor, een 8085, die middels de zg microsoftware, dit is een programma in EPROM op het CMC board, de commando's die in het buffer gezet worden interpreteert. Aan de hand daarvan kan hij allerlei I/O poorten aansturen die dan zorgen voor de uiteindelijke instellingen van de hardware debugger, zoals triggerwoorden, qualifiers, eventcounter etc. Ook kan hij de target processor bepaalde instructies laten uitvoeren waardoor deze bv. zijn registers laadt met bepaalde waarden, of juist zijn registerinhouden prijsgeeft.

Nadat een emulatie gestopt is zullen allerlei gegevens die tijdens de emulatie ingenomen zijn moeten worden doorgegeven aan de P851. Ook deze taak wordt, in opdracht van de debugprocessor, verricht door de 8085. De gegevens worden daarna door de debugprocessor geschikt gemaakt voor presentatie op het scherm.

Via de CMC-bus (voor commando's) en de IDB (Internal Debug Bus) met voornamelijk data, is het CMC-board verbonden met het "Trace Board". Hierop zijn aanwezig het RTTM met zijn trigger-, qualifier- en clock logica en de event counter met zijn start-, stop- en clock logica.

Middels dezelfde bussen is er ook een verbinding met het "Real Time Board". Dit is het hart van de HardWare Debugger. Alle start- en stop condities zijn hierop verzameld. Hierop komen ook indirect alle signalen van de targetmicroprocessor en de userprobes binnen en zitten de triggerwoorden en qualifiers, zodat hier bepaald en gestuurd kan worden wanneer een emulatie start en stopt. Hierop bevindt zich ook een buffergeheugen dat toegankelijk is voor zowel de targetprocessor als de CMC, zodat emulatiegegevens en opdrachten voor de targetprocessor via dit geheugen kunnen worden uitgewisseld.

PHILIPS

Tenslotte is nog aanwezig de mapping logica: De acht meest significante adreslijnen van de targetprocessor worden via een door de CMC instelbaar 256x8 ram gemapped naar een 1 Mbyte geheugenruimte. Bovendien wordt via een 256x3 ram aangegeven of een geheugenblok guarded is, of het al dan niet write protected (rom) is en of het "in MDS" is of niet. Met dit laatste bit worden ook de buffers gestuurd die zorgen dat de targetprocessor zijn data uitwisselt met het juiste geheugen (emulatie- of prototype geheugen). De Input- en Output map bestaan ieder uit een 256x1 ram, dat bepaalt of een poort of een blok van poorten al dan niet in het prototype aanwezig is.

Tot zover was de hardware universeel, dwz. onafhankelijk van het type targetprocessor. Aangesloten op het RTB maar buiten de MDS kast hebben we nog de MAB (Microprocessor Adaptor Box). Uit dit kastje komen komen de lijnen waaraan de probe zit die in het prototype de targetprocessor vervangt. Op de MAB komt in een speciale socket de targetprocessor (Als nog geen prototype aanwezig is komt de targetprocessor in deze socket en de probe in een tweede identieke socket op de MAB).

Via BNC connectors kunnen we de twee triggersignalen afnemen en een externe clock toevoegen. 1 Userprobe wordt in geval van een 8 bits targetprocessor via de MAB aangesloten, de andere komt aan het CMC board.

De eigenlijke taak van de MAB is het aanpassen van de specifieke signalen van de targetprocessor naar de universele IDB (internal debug bus) en het bufferen van die signalen. Het zal duidelijk zijn dat voor elk type target processor een andere MAB nodig is.

MULTI PROCESSING

NAME NAAM Han v.d.Broek | SUPERS. VERV. | SH. BL. | SH. BL. 2 — 14

PROPERTY OF EIGENDOM VAN N.V. PHILIPS' GLOEILAMPENFABRIEKEN-EINDHOVEN-NEDERLAND | CHECK CONTR. | DAT. | FORM. A4

III ONTWIKKELING VAN GEBRUIKERS SOFTWARE

3.1 Debugstrategie voor multi-processor systemen

Als u een multi-processor systeem wilt ontwerpen zult u waarschijnlijk als volgt te werk gaan:

U hebt een software team, dat zal beginnen met het ontwikkelen van de software voor elk processor systeem afzonderlijk als betroffen het een aantal onafhankelijke uni-processor systemen. Hierbij kunnen de mensen dan met vrucht gebruik maken van het MDS zoals dat in het vorige hoofdstuk is beschreven, om de programma's te schrijven en te debuggen.

Parallel hieraan zal een hardware team beginnen aan het ontwikkelen en later testen van de afzonderlijke hardware modules. Belangrijke instrumenten voor hen om de systeempjes te debuggen zijn o.a. oscilloscoop, en logic timing analyzer. Ook erg makkelijk hierbij is het om kleine testprogramma's te maken. Deze kunnen gemakkelijk met een MDS geschreven worden en ook via het MDS op de hardware draaien. Een logic state analyzer kan dan aantonen of de programmaatjes ook werkelijk goed werken.

Als zowel hard- als software per systeem goed lijken te werken kunnen we ze met het MDS gaan integreren om zodoende te komen tot een aantal goed werkende, op zichzelf staande, microprocessor systemen.

Dan wordt het voor het software team tijd om de routines te maken die gebruik maken van de interactie tussen de processoren en de routines die deze interactie verzorgen. Om deze zaken te debuggen moet gebruik gemaakt kunnen worden van een MDS dat hiervoor is ingericht. Aan wat er bij het huidige PMDS nog ontbreekt, en hoe dit gerealiseerd kan worden, om multiprocessor software te debuggen is dit hoofdstuk gewijd.

Intussen zal het hardware team de verschillende processorschakelingen met elkaar gaan verbinden. Dit houdt in dat ze de bus(sen), arbitrage- en prioriteiten logica en interrupt mechanismen gaan bouwen. Ook deze zaken zullen getest moeten worden, waarbij dan logic analyzers, in combinatie met testprogramma's goede diensten kunnen bewijzen. Toch zal het makkelijk zijn als er apparaten beschikbaar zijn waarmee een aantal specifieke dingen eenvoudig getest kunnen worden. Ik denk hierbij aan "mutual exclusion", het testen van buffers en geheugen modules en dergelijke. Hoofdstuk vier behandelt deze punten als uitbreidingen van het PMDS.

De laatste fase van de ontwikkeling is dan het met elkaar integreren van de totale hard- en software. Ook hierbij zal een MDS voorzien van de nodige uitbreidingen goede diensten kunnen bewijzen.

3.2 Ondersteuning van communicatie op I/O basis

In het eerste hoofdstuk hebben we gezien dat een manier om twee processoren met elkaar te laten communiceren, is dat de ene processor data in een bepaalde I/O-poort of queue schrijft, hetwelk dan door de ander gelezen kan worden. Zo'n poort of queue hoeft vanuit beide microprocessoren gezien niet hetzelfde adres te hebben. Om een dergelijke communicatie te steunen moeten er binnen het MDS een aantal poorten cq queues zijn die toegankelijk zijn voor meerdere target processoren. We zullen nu bekijken hoe we dit kunnen realiseren; hierbij beschouw ik enkelvoudige poort als een bizonder geval van een queue, nl een queue met diepte een.

We dienen nu een mogelijkheid te creeren in het MDS waarmee een aantal processoren op deze manier kunnen communiceren. We beschikken al over de I/O mapping waarmee we kunnen aangeven of een bepaalde I/O poort in het prototype aanwezig is. Als we die nu in dit geval zo instellen, dat de poorten die voor onderlinge communicatie bedoeld zijn niet in het prototype zitten, dan zal de emulatie stoppen op het moment dat zo'n poort wordt aangesproken, dus als er communicatie gepleegd gaat worden.

Als de emulatie eenmaal stilstaat kan de debuggersoftware de data ophalen die de zendende processor weg wilde schrijven in een buffer. Deze zou dan tijdelijk opgeslagen moeten worden in een bepaalde queue die voor dit doel in de werkruimte van de P851 moet kunnen worden geschapen. Zodra een tweede processor de betreffende data wil lezen zal de emulatie weer stoppen t.g.v. de instelling van de I/O map. De debugprocessor moet dan die data uit de queue halen en doorgeven aan de processor die ze wil hebben. Telkens na interventie van de debugprocessor kan de emulatie automatisch doorgestart worden. De communicatie is hiermee dan een feit geworden.

Het grote voordeel van deze methode is dat er geen extra hardware nodig is om de oplossing te realiseren. Het real time karakter van de emulatie gaat op deze maniet wel verloren. Bovendien kan de onderlinge synchroniteit van de verschillende microprocessoren erg scheef getrokken worden. Dit laatste kunnen we ondervangen door in geval van een emulatiestop alle geemuleerde processoren te laten stoppen middels de XEEMU lijnen, die hier speciaal voor zijn.

Omdat we hier praten over het testen van software is deze vertraging geen bezwaar, mits de emulatie niet hinderlijk lang gaat duren. Gezien de aard van het datatransport (via buffers van beperkte lengte) mogen we verwachten dat de mate van datauitwisseling zodanig beperkt zal zijn dat telkens stoppen van de emulatie bij uitwisseling niet zal leiden tot onaanvaardbaar lange emulatiesessies. Tests waarbij het 'real time' karakter niet verloren mag gaan zijn alleen noodzakelijk, ten eerste als er timing kritische hardware aanwezig is. In zo'n geval mogen we verwachten dat ook de noodzakelijke I/O buffers in het systeem aanwezig zijn, zodat simulatie niet nodig is.

Een tweede geval waarin het niet "real time" aspekt een bezwaar kan zijn is als we (een) processor(en) willen toevoegen aan een bestaand systeem met timing kritische aspekten. Ik geloof echter dat we er vanuit mogen gaan dat in de omgevingen waarin een MDS gebruikt zal worden ook een target systeem aanwezig is dat (eventueel tijdelijk) voor debug doeleinden kan worden gebruikt, in welk geval de timing kritische aspekten geen rol spelen.

Een gebruiker zal een buffer voornamelijk op twee manieren kunnen implementeren: In het ene geval als FIFO, de data die er het eerst worden ingezet, worden ook weer het eerst gelezen. In het andere geval worden de data die laatst zijn geschreven het eerste uitgelezen (stack principe).

Software fouten die in een gebruikersprogramma kunnen optreden zijn dat de zendende processor data probeert te schrijven in een vol buffer: Als de communicatie tussen de processoren in de software niet goed gesynchroniseerd is zou de zender een tweede bericht kunnen schrijven, voordat het eerste gelezen is.

Indien het betreffende buffer een gecombineerd zend- ontvang buffer is, kan het voorkomen dat meerdere processoren gelijktijdig proberen toegang te krijgen. De debugprocessor zou dit soort fouten kunnen detekteren en een overeenkomstige foutmeldingen kunnen genereren.

Als we met het maximaal mogelijke aantal van vier target processoren een volledig verbonden systeem maken zijn er twaalf buffers nodig. Nl er zijn zes mogelijke onderlinge verbindingen. Iedere verbinding kan bestaan uit een zend- en een ontvang buffer. Voor deze toepassing kunnen we dus volstaan met twaalf in de debugprocessor gedefinieerde queues. De diept ervan is vrij arbitrair. Het is erg moeilijk te voorspellen hoe diep de toekomstige gebruikers hun buffers zullen maken. Ik denk zelf aan waarden tussen de 64 en 256 woorden. Het zal echter in de prakrijk nauwelijks voorkomen dat iemand ze alle 12 nodig heeft tijdens een emulatie, dus een aantal van acht mag voldoende geacht worden.

Concluderend kunnen we zeggen dat om communicatie op I/O basis te ondersteunen geen extra hardware nodig is. Wel zal de debugprocesssor een aantal toevoegingen nodig hebben:

- Een I/O poort die is gedefinieerd "in MDS" zal behalve de status "guarded", "console", of "diskfile" ook voor communicatie geschikt moeten zijn.
- Er moeten maximaal 12 buffers in de werkruimte van de debugprocessor gedefinieerd kunnen worden, 8 is waarschijnlijk voldoende.
- De gebruiker moet kunnen opgeven welke I/O adressen van verschillende targetprocessoren hetzelfde buffer aanduiden.
- De gebruiker moet als hij dit wenst een maximale bufferlengte kunnen opgeven. Indien een programma cq processor probeert meer data in dit buffer te schrijven moet dit resulteren in een foutmelding.
- De andere genoemde gebruikersfouten kunnen slechts gedetekteerd worden als de lengte van een bericht bekend is. Dit zal vaak een vaste lengte zijn, of de lengte wordt meegedeeld in het eerste woord van het bericht. We zouden kunnen besluiten om voor deze gevallen een meer uitgebreide foutendiagnose in de debugprocessor in te bouwen.Dit wordt dan wel vrij moeilijk en is weinig universeel toepasbaar.

3.3 Communicatie via gemeenschappelijk geheugen.

Een tweede mogelijkheid voor processoren om te communiceren is via een gemeenschappelijk geheugen. Zoals we in hoofdstuk I gezien hebben kunnen de processoren op meerdere manieren met de geheugens worden verbonden. Op welke manier dat gebeurt is hier niet van belang, essentieel is echter dat er een of meerdere stukken geheugen zijn die toegankelijk zijn voor meerdere processoren. Als we tijdens de software ontwikkelingsfase de communicatie van via dit principe werkende systemen willen emuleren zullen we in staat moeten zijn de targetprocessoren toegang te geven tot de data in gemeenschappelijk geheugen. Door de gekozen opzet van de hardware debuggers is dit nu niet mogelijk. Immers ze hebben allemaal alleen toegang tot hun eigen emulatiegeheugen en ze kunnen ook niet bv. werken in het systeemgeheugen.

We kunnen in dit geval een oplossing bieden in de hardware en in de software. Ook is een gecombineerde oplossing theoretisch mogelijk. Ik zal deze drie mogelijkheden nu beschrijven.

3.3.1 Software oplossing

De communicatie gaat bij deze oplossing als volgt: Het Het (emulatie)geheugen van een van de targetprocessoren zeg A, bevat de data die later, in de uiteindelijke configuratie van de gebruiker in het gemeenschappelijke geheugen komen te te staan. Als een andere targetprocessor, zeg B, toegang wil tot deze data zal de emulatie gestopt moeten worden, de debugprocessor zal moeten zorgen dat de data uit het juiste geheugen gehaald worden (in geval van een lees operatie), in dit geval dus uit geheugen A en deze data zullen doorgespeeld moeten worden naar processor B, waarna de emulatie doorgestart kan worden. Bij een schrijfoperatie gaat het natuurlijk precies omgekeerd.

Allereerst moeten we zorgen dat de emulatie gestopt wordt. Dit kunnen we bv doen door in de memory maps het bit "guarded" van het betreffende geheugenblok actief te maken. Zodra nu een adres uit het gemeenschappelijk geheugen wordt aangesproken stopt de emulatie. (Omwille van de synchronisatie laten we in zo'n geval alle processoren stoppen middels de XEEMU-lijnen.)

We moeten daarna aan de weet zien te komen welke operatie er op welke geheugen inhoud moet worden uitgevoerd. Het grote probleem is hier nl. dat we een emulatie pas kunnen stoppen nadat een instruktie echt is uitgevoerd. Zo zit de hardware debugger nu eenmaal in elkaar. In dit geval moeten we dus de target processor zijn instruktie (eventueel foutief) laten uitvoeren. Als de emulatie eenmaal gestopt is moeten we nagaan welke instruktie is uitgevoerd en deze nogmaals, maar nu wel correct, laten uitvoeren. Er is hier een maar: Als nl. t.g.v. de instruktie bv. het adres waar de data stond veranderd wordt (zoals bv. bij MOV H,M) dan kunnem we de juiste instructie nooit meer achterhalen.

Stel dat de inhoud van geheugenplaats M in register R van processor P2 moet worden geplaatst, waarbij M in het (emulatie)geheugen van P1 ligt. De debugprocessor moet dan zorgen dat P1 de inhoud van M leest. Als dit gebeurd is heeft de debugprocessor toegang tot de inhoud en kan deze in register R van P2 zetten.

Als beide processoren weer in de staat zijn gebracht van voor de emulatiestop (alle registers de oude waarde), behalve dat P2 een andere waarde in R heeft kan de emulatie worden doorgestart. Al deze handelingen kunnen al door de bestaande oftware worden verricht.

Bij een volgende instructie moet wellicht de inhoud van geheugenplaats N in register S komen. We zien hieruit dat de debugprocessor precies moet weten welke instructie er uitgevoerd is alvorens hij de emulatie kan doorstarten. Het is dus noodzakelijk dat de debugprocessor van elk type targetprocessor alle instructies kent die op geheugen werken om adequaat de data te kunnen verwerken. Bovendien moet in de werkruimte van de debugprocessor een afbeelding aanwezig zijn van de memory maps van de verschillende hardware debuggers, voor zover ze betrekking hebben op gemeenschappelijk geheugen. Er moet nl bepaald kunnen worden welke targetprocessor we moeten inschakelen om onder welk adres data op te halen resp. weg te schrijven als er een processor toegang verlangt tot gemeenschappelijk geheugen.

Momenteel is het zo dat de monitor (dit is het voornaamste besturingsprogramma van het MDS, dat altijd in het systeemgeheugen aanwezig moet zijn) en de debugprocessor niet samen in het werkgeheugen passen. De debugprocessor beslaat nl. alleen al meer dan 80 k. Dit heeft tot gevolg dat voor practisch elke actie van de HWD eerst een bepaalde "overlay" van de debugprocessor van de diskette gehaald moet worden. In ons geval zal voor het transporteren van een datawoord drie tot vier keer een diskacces nodig zijn nl. voor het stoppen van de emulatie en het uitlezen van de status, voor de command interpreter, voor de memorymap en voor het doorstarten van de emulatie. Een en ander heeft tot gevolg dat zo'n transactie meer dan een seconde gaat duren. Bedenken we dan bovendien dat communicatie op basis van gemeenschappelijk geheugen vrijwel alleen toegepast zal worden als deze communicatie vrij intensief is, dan kunt u zich voorstellen dat een dergelijke simulatie snel zal leiden tot onaanvaardbaar lange debugsessies.

Deze problemen kunnen worden omzeild door met een hard disk te werken, echter ook zonder hard disk moet het systeem acceptabel kunnen werken. Een tweede alternatief zou zijn meer werkgeheugen. Echter ook de opvolger van de P851 zal waarschijnlijk niet meer dan 64 k aankunnen en bovendien vindt ik een minimumsysteem met 64 k al vrij veel bij de huidige prijzen van de geheugens.

Om deze reden en vanwege de niet onaanzienlijke hoeveelheid extra software die nodig zou zijn vind ik deze methode weinig aantrekkelijk.

3.3.2 gemengde oplossing

Er is een tweede methode mogelijk, waarbij de inhoud van geheugenlocaties via de debugprocessor wordt doorgegeven van de ene hardware debugger naar de andere. Deze is sneller dan de vorige maar vergt bovendien extra hardware. Hij berust op het volgende principe:

Wederom is het gemeenschappelijke geheugen ondergebracht in het emulatiegeheugen van 1 van de targetprocessoren. zodra een microprocessor een geheugenlocatie aanspreekt die later als gemeenschappelijk zal fungeren en die niet in zijn (emulatie)geheugen aanwezig is zouden we die processor 'not ready' kunnen maken door zijn ready-ingang te activeren. De adressen blijven dan op de bus staan en we kunnen zien of de instruktie een lees- of schrijfoperatie verlangt. In geval van een schrijfoperatie staan de data op de databus klaar. Als nu de debugprocessor in staat is deze gegevens (adres, data en lees/schrijf commando) over te nemen en hij weer over eerder genoemde memory map beschikt kan hij de betreffende data op de juiste plaats uitlezen resp. wegschrijven. Zodra de data is weggeschreven of, bij een leesoperatie is opgedrukt op de databus van de lezende target processor, kan de emulatie worden doorgestart door de ready ingang inactief te maken.

Deze methode is sneller dan de vorige omdat we nu niet de status van de verschillende targetprocessoren hoeven te lezen en terug te schrijven en omdat we minder diskaccessen nodig hebben. De extra's die we nodig hebben zijn de volgende:

- Opdat de readylijn snel geactiveerd kan worden moet de memorymap in de HWD met een bit per geheugenblok worden uitgebreid. Dit bit moet aangeven of het betreffende geheugenblok al dan niet gemeenschappelijk geheugen is, dat buiten de HWD aanwezig is.
- Als tijdens de emulatie dit bit geactiveerd is door een acces tot een geheugenplaats uit zo'n blok moet dit ogenblikkelijk aan de MAB worden doorgegeven, zodat deze de targetprocessor snel 'not ready' kan maken.
- Ook moeten de andere targetprocessoren tijdelijk gestopt worden. Dit kan door het extra bit te verbinden met de stuurlogica voor de XEEMU-lijn.
- Via extra poorten zal de CMC in staat worden gesteld om de adreslijnen te lezen en de datalijnen te lezen en te beschrijven en om de status (lees of schrijf) te bekijken.
- Dan moet de software nog de 'not ready' status kunnen opheffen. Hiervoor zijn nog een paar poortjes op het Real Time Board nodig.

Samenvattend kunnen we nu stellen dat er op het RTB wat extra hardware moet komen. De debugprocessor moet nu de memorymap kunnen bijhouden, de transfer kunnen laten plaatsvinden en de uitgebreide memorymap kunnen instellen. Op deze manier kan de transfer sneller gaan, vooral omdat er minder diskacces nodig is dan in het vorige geval.
Het grote probleem hierbij op het ogenblik is dat als een processor not ready gemaakt wordt al de genoemde gegevens op de IDB (Internal Debug Bus) staan. Om echter te kunnen communiceren met de debugprocessor moet de CMC (Control Micro Computer, de 8085) ingeschakeld worden en deze maakt ook gebruik van de IDB. Tenzij we de hele structuur willen veranderen is deze oplossing dus vooralsnog onmogelijk.

Tot slot nog een algemene opmerking met betrekking tot simulaties. Als tijdens een simulatie de targetprocessor commando's moet uitvoeren onder opdracht van de CMC moet tijdens die commando's het tracememory, de eventcounter en de triggerlogica tijdelijk disabled worden, zodat we bij een definitieve emulatiestop alleen de gegevens zien van de gebruikersprogramma's.

3.3.3 Hardware oplossing

Als we communicatie via gemeenschappelijk geheugen in hardware mogelijk willen maken betekent dit noodzakelijkerwijs dat alle hardware debuggers moeten worden aangesloten op een "gemeenschappelijke emulatiebus". We kunnen ons samenstel van maximaal vier HWD's in dit opzicht dan beschouwen als een multiprocessor systeem. Uit hoofdstuk I is bekend dat het totale systeemgeheugen (hier dus emulatiegeheugen) op een aantal manieren aan de verschillende microcessoren (lees HWD's) kan worden gekoppeld.
Omdat het PMDS op het ogenblik vanuit de software gezien vier debuggers aankan en omdat dit ook in alle publicaties wordt vermeld zal ik me alleen bezig houden met oplossingen die geschikt zijn om de betreffende communicatie inderdaad voor vier targetprocessoren gelijkertijd mogelijk te maken. Dat dit gelijktijdig emuleren van vier processoren waarschijnlijk praktisch niet zo zinvol zal zijn en dat je andere eisen kunt gaan stellen als je oplossingen bekijkt die bv. geschikt zijn voor een systeem met maar twee debuggers zal ik hier buiten beschouwing laten.
Om tot een bepaalde keuze te komen zal ik eerst een aantal eisen noemen waaraan onze configuratie naar mijn mening dient te voldoen:

PHILIPS

1- De bestaande emulatiegeheugenkaarten (2 soorten) moeten zo min mogelijk en liefst helemaal niet gewijzigd hoeven worden.
2- De emulatie moet zoveel mogelijk 'real time' verlopen, dwz. de snelheid moet zo dicht mogelijk liggen bij de snelheid die de gebruiker later in zijn prototype zal halen.
3- We moeten het systeem zonder veranderingen aan te brengen zowel voor multi-ICE als voor single-ICE kunnen gebruiken. (het laatste natuurlijk door tijdelijk maar 1 van de aanwezige HWD's te gebruiken.)

Ad 1)

Als we onze geheugenkaarten niet willen wijzigen verwerpen we daarmee de mogelijkheid om een multiport systeem te maken. Nl voor een multiport systeem moet er arbitrage bij de geheugens aanwezig zijn. Deze logica zet je normaliter natuurlijk het liefst op de geheugenkaarten, maar dat willen we hier niet. We zouden ook een extra kaart kunnen maken waarop voor alle geheugens de arbitrage aanwezig is. Dit heeft als grote nadeel dat we voor elke emulatiegeheugenkaart die we in het systeem willen toelaten arbitrage en buffers voor 4 ingangen (de 4 HWD's) moeten maken. In het algemeen zullen we dan veel te veel maken omdat lang niet alle gebruikers het maximum systeem zullen kopen. Bovendien is deze logica van nature gecompliceerd en dus duur.

Ad 2)

Bij multiprocessorsytemen kan er nooit sprake zijn van een echte 'real time' situatie. Immers het is niet voorspelbaar hoe lang een processor moet wachten om toegang te krijgen tot een bepaalde eenheid, als hij daarbij gebruik maakt van een gemeenschappelijke faciliteit. Als er een aantal gegevens bekend zijn over het prototype, zoals oa, aantal microprocessors, toegangstijd tot gemeenschappelijk geheugen, tijd nodig voor arbitrageslagen etc, en over een programma (verhouding gebruik prive en gezamenlijk geheugen e.d.) kunnen we wel een verwachtingswaarde geven voor de duur van programma's. Een multiprocessor emulatie zouden we dan 'real time' kunnen noemen als tijdens de emulatie de verwachtingswaarde binnen bepaalde grenzen bereikt wordt.

Als we een systeem zouden maken waarbij al het emulatie geheugen gemeenschappelijk is zal een emulatie op zo'n systeem zeker langer duren dan de praktijksituatie, als daarin ook prive geheugen aanwezig is bij elke processor. M.a.w. we moeten een systeem ontwerpen waarin zowel ruimte is voor prive- als voor gemeenschappelijk geheugen.

MULTI PROCESSING

NAME NAAM Han v.d.Broek | SUPERS. VERV. | SH. BL. | SH. BL. 3 — 9

PROPERTY OF EIGENDOM VAN N.V. PHILIPS' GLOEILAMPENFABRIEKEN·EINDHOVEN–NEDERLAND | CHECK CONTR. | DAT. | FORM. A4

Ad 3)

Aan deze eis is waarschijnlijk het eenvoudigst te voldoen mits we een systeem ontwerpen waarbij de bestaande emulatie-geheugen kaarten ook gebruikt kunnen worden, zodat de gebruiker niet afzonderlijke kaarten nodig heeft voor single- en multi ICE. Bij gemeenschappelijke geheugens hebben we altijd bufferlogica nodig. Omdat deze nu al op de bestaande emulatiegeheugen kaarten aanwezig is, zal dit aspekt niets afdoen aan het real time karakter van de emulatie in geval van single ICE.

3.3.4 Indeling van de extra debuggerkast

Aan extra logica hebben we nu nodig:

-arbiters (max 4) om te bepalen welke HWD's toegang vragen
Aan extra logica hebben we nu nodig:
tot gemeenschappelijk geheugen en om te zorgen dat ze ook inderdaad toegang krijgen als blijkt dat aan een bepaalde prioriteits voorwaarde is voldaan.
- prioriteiten logica om bij meerdere gelijktijdige aanvragen te beslissen welke HWD echt toegang krijgt.
- logica om te zorgen dat het acknowledge signaal TSMN van de emulatiegeheugens naar de juiste HWD wordt doorgegeven.
- logica die bij elke geheugenaanvraag bepaalt of dit stuk geheugen prive is of gemeenschappelijk, in welk laatste geval de arbiter moet worden aangesproken.

Het moet mogelijk zijn de drie eerst genoemde stukken logica op een kaart van het dubbele EURO formaat te bouwen. Deze kaart zal in het vervolg met de naam arbiter kaart worden aangeduid.
Zoals bekend zullen extra HWD's in een losse kast geplaatst worden. Om u een indruk te geven van het systeem zoals mij dat voor ogen staat is een mogelijke indeling van deze kast getekend (zie fig 1).

We kunnen in deze figuur een aantal punten onderscheiden:

De "expa".

We hebben twee mogelijkheden voor de verbinding tussen mainframe en extra kast:
1) We kunnen de UPL-bus naar buiten uitvoeren. Vanwege de specificaties van deze bus kan dit niet zonder meer. Er moet dus een interface komen in het mainframe waarop de UPL-bus netjes wordt afgesloten en met drivers en receivers voor een flatcable waaraan in de extra kast een "expander" zit met drivers en receivers voor een UPL-achtige bus. (Onder UPL-achtige bus versta ik in dit geval een bus met dezelfde signalen, maar eventueel een andere timing dan de UPL-bus).
De logica die een en ander moet verzorgen bestaat reeds in andere computersystemen. Dit zijn twee kaarten t.w. de "IOTZ" moet in het mainframe komen, de "expander" zou dan in de debuggerkast geplaatst moeten worden. Deze kaarten kunnen zonder meer in ons systeem toegepast worden. Ze zijn weliswaar duur, maar vergen geen extra ontwikkelkosten.
In de extra kast kan een groot connector paneel komen dat onderverdeeld moet worden in verschillende bussen. Een daarvan is een UPL-achtige bus met 4 slots:
- 1 voor de "expa"
- 2 voor de CMC-boards van 2 verschillende HWD's
- 1 voor een flatcable verbinding met een tweede UPL-achtige bus voor de laatste HWD. Dit laatste slot vergt geen extra kastruimte omdat het achter een trace-board kan vallen.

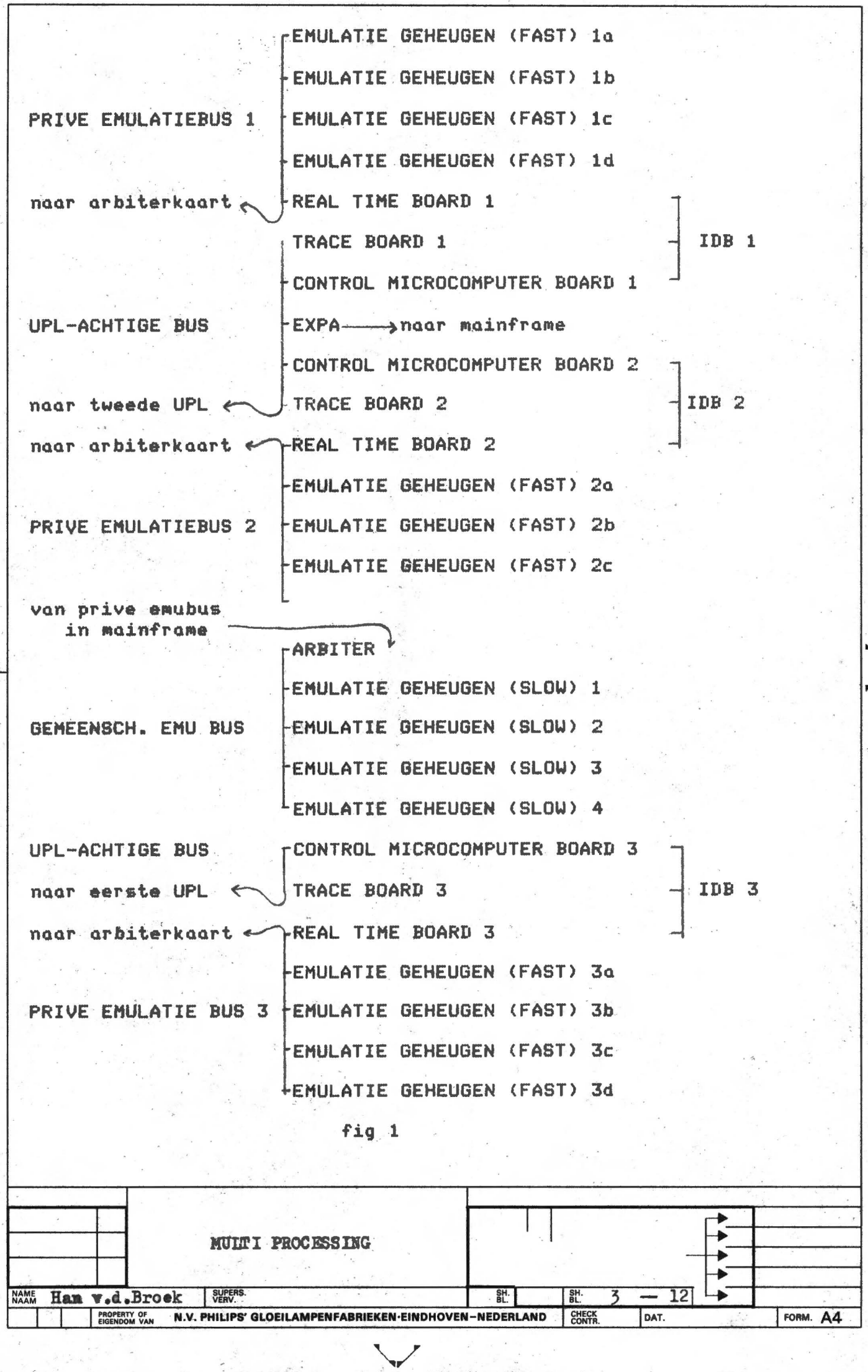


fig 1

MULTI PROCESSING

NAME NAAM	Han v.d.Broek	SUPERS. VERV.	SH. BL.	SH. BL. 3 — 12	
PROPERTY OF EIGENDOM VAN	N.V. PHILIPS' GLOEILAMPENFABRIEKEN-EINDHOVEN-NEDERLAND		CHECK CONTR.	DAT.	FORM. A4

2) We kunnen de verbinding ook op seriele basis maken. Dan zijn geen expanders nodig en UPL-achtige bussen en kunnen we met 1 slot minder toe. Wel moeten in het mainframe een-, twee-, of drie voudige seriele interfaces komen en de CMC-boards worden voorzien van een seriele aansluiting i.p.v. de huidige I/O poorten. Een drievoudige seriele interface is op het ogenblik in ontwikkeling en zal zeker op korte termijn beschikbaar komen. Wel moeten de CMC boards omgebouwd worden. De mogelijkheid hiervan is reeds sumier onderzocht.

Ik verwacht dat de hardware kosten voor deze oplossing iets gunstiger liggen, echter het verschil in snelheid maakt hem veel ongunstiger.

Per HWD is er een emulatiebus

Als we het standpunt handhaven dat we 64 k fast emulationmemory per HWD mogelijk moeten maken zal zo'n emulatiebus 6 slots moeten omvatten nl - 1 voor het RTB

- 4 voor de geheugenkaarten
- 1 voor een flatcable verbinding met de arbiterkaart.

Deze flatcable aansluiting kan soms achter het trace-board vallen, zodat hij geen extra kastruimte vraagt.

Er is een "gemeenschappelijke emulatiebus"

Hieraan zijn de slow emulation memories (fast heeft hier weinig zin) aangesloten, en ook de arbiterkaart. Een slow emulation memory kan 64 k groot zijn, dus met 4 kaarten voor het gemeenschappelijk geheugen moeten we kunnen volstaan. Hiermee komt deze bus op 5 slots.

De arbiter zal via flatcables moeten worden verbonden met de diverse prive emulatie bussen (dus ook met die in het mainframe). Hier moeten we dus een emulatiegeheugen kaart opopofferen ten bate van de genoemde verbinding. Een alternatief is nog dat we de emulatiebussen in de hoogte uitbreiden met een connector waaraan de verbinding met de arbiter gelegd kan worden. Deze mogelijkheid is alleen reeel als we op het connectorpaneel alle connectors aanbrengen, dus twee rijen boven elkaar. Dit zal wel problemen geven met de koeling.

Er is een drietal IDB's nodig.

Nl een per HWD met 3 slots voor CMC, RTB en het traceboard.

Al met al komen we op deze manier op een extra kast die, schrik niet, 27 kaarten zal moeten kunnen bevatten plus 2 of 3 voedingseenheden. Vijf slots komen voor rekening van de gemeenschappelijke emulatiebus. Zij zijn niet aanwezig als we kiezen voor een software oplossing van ons communicatieprobleem, hoewel we ons moeten afvragen of we in dat geval de prive emulatie bussen niet moeten uitbreiden.

- Dit zou het aantal slots voor de extra HWD-kast terug brengen tot 20. Dit is een aantal dat is onder te brengen in een standaard 19 inch rek (incl voeding)
- We moeten ons afvragen of het zinvol is om meer dan twee processoren gelijktijdig te emuleren. Immers als we een bepaald stuk communicatie willen bekijken kunnen we de programma's zodanig in stukken verdelen dat we ieder stuk op zich kunnen debuggen en dat ieder stuk maar gebruik maakt van maximaal 2 processoren.
- De debugprocessor, die toch al overvol zit wordt er iets eenvoudiger door. Zeker als we rekening houden met al gedane en nog volgende voorstellen met betrekking tot de debugprocessor.

3.3.5 Memory mapping

De arbiter zal moeten weten of een adres dat op de emulatiebus staat geheugen aanspreekt dat prive is, of dat dit gemeenschappelijk is. M.a.w. of de gemeenschappelijke bus moet worden aangevraagd of niet.

De eenvoudigste manier om dit te bepalen zou zijn voor elke target processor de prive adressen te mappen naar een bepaalde adresruimte, zeg de eerste 64 k. Dan kunnen we met 1 TTL IC bepalen of een adres prive- dan wel gemeenschappelijk geheugen betreft. Op deze manier is het niet mogelijk de prive adresruimte groter dan 64 k te definieren, hetgeen wel eens nodig kan zijn. Door de mogelijke prive ruimte groter te definieren beperken we de gemeenschappelijke ruimte weer te zeer. Daarom moeten we zoeken naar een andere oplossing.

We kunnen de memorymap uitbreiden met 1 bit per blok. Tijdens de fase van het opgeven van de memorymap zouden we dan van elk blok op moeten geven wat voor type het betreft (gemeenschappelijk of prive geheugen). Dit bit kunnen we rechtstreeks gebruiken om de arbiter een aanvraag te laten doen voor gemeenschappelijk geheugen als het "common" aangeeft. Door als defaultwaarde voor dit bit de waarde "private" te kiezen bereiken we dat in gevallen van single ICE er voor de gebruiker niets verandert.

Door de vele mogelijke systeemconfiguraties wordt het voor de gebruiker lastig om bij te houden hoeveel emulatiegeheugen er bij de verschillende HWD's als private aanwezig is en hoeveel in het hele systeem als gemeenschappelijk, wat er slow en wat er fast is, op welke adressen dit allemaal aanwezig is en op welke slots de verschillende kaarten aanwezig zijn. Hier kan de software hulp bieden. We zouden een routine kunnen maken die na intypen van het commando HELP o.i.d. gaat testen welke blokken emulatiegeheugen er bij elke HWD aanwezig zijn, en welke blokken gemeenschappelijk aanwezig zijn. Mededelingen hieromtrent kunnen dan op het scherm verschijnen.

We hebben niet de mogelijkheid om te zien of geheugen fast of slow is. Indien gewenst zouden de kaarten gewijzigd kunnen worden, zodanig dat dit eenvoudig getest kan worden.

I.p.v. of naast de "HELP" functie zouden we met symbolische adressen kunnen werken. De debugprocessor moet dan zelf deze symbolische adressen vertalen naar adressen van aanwezig geheugen. Praktisch houdt dit in dat dan na het laden van de debugprocessor meteen de "helptest" moet worden uitgevoerd. Er moet ruimte komen voor de symbolische memorymap We moeten kunnen opvragen hoe de mapping is uitgevoerd. En we moeten in staat zijn om ook nu een aantal parameters op te geven zoals "guarded", write protected", "in MDS" en rekening kunnen houden met fast en slow emulatie geheugen.

3.3.6 Arbitrage en prioriteits bepaling

De arbitrage kan het eenvoudigst op een seriele manier plaats vinden zoals beschreven in par 1.5 . Ik heb de signaalbenamingen en de functies zo gekozen als bij de geintegreerde busarbiter van Intel, de 8289. Deze vraagt echter als inputs een aantal statussignalen van de 8085 of 8086. Daar wij deze niet ter beschikking hebben is dit IC voor ons niet bruikbaar. Dat een arbiter toch vrij eenvoudig te bouwen is moge blijken uit bijlage I.

Ter bepaling van de prioriteit is de seriele manier gekozen omdat deze het eenvoudigste en het goedkoopste te maken is. Hierdoor is het mogelijk dat de processor met de laagste prioriteit weinig aan bod komt, waardoor de onderlinge synchroniteit t.o.v. de werkelijke situatie verloren kan gaan. Dit zal echter bij elke vorm van prioriteitsbepaling het geval zijn, en zelfs in de werkelijke situatie zal deze telkens anders liggen.

3.4 Acknowledge signalen

Zoals bekend is het vooral bij systemen waarin een geheugenacces een onbepaalde tijd kan duren, bv multiprocessorsystemen, van belang dat we de microprocessor steeds synchroniseren met de momentane accestijd. De eenvoudigste en veel toegepaste methode (bv op Intel single board computers) om dit te bereiken is een "ready circuit" te maken zodanig, dat de microprocessor altijd in de toestand "not ready" verkeert. Alle eenheden die door de processor geadresseerd kunnen worden , zoals (p)rom, prive- en gemeenschappelijk geheugen, I/O devices, hebben dan de verantwoordelijkheid om te zorgen dat de microprocessor tijdelijk "ready" wordt als ze geadresseerd zijn en hun data "valid" zijn.

Als we zo'n systeem willen emuleren, waarbij dan een deel van het geheugen bestaat uit emulatiegeheugen in het MDS, moeten we er voor zorgen dat als het emulatiegeheugen wordt aangesproken er ook door het MDS een acknowledge signaal wordt doorgegeven aan de targetprocessor. Om dit te bereiken kunnen we zorgen dat altijd als het emulatiegeheugen wordt aangesproken het acknowledge signaal TSMN een eventuele "not ready" status opheft. Dit vergt twee poortjes op het real time board.

3.5 Kritische secties

Nauw samenhangend met communicatie via gemeenschappelijk geheugen is het gebruik van kritische secties. Een MDS is het ideale hulpmiddel om deze te bewaken tijdens de fase van het testen van de software. Het is helemaal niet ondenkbaar dat een programma ongewenst een kritische sectie betreedt Bv. omdat men vergeten is een semafoor te testen of te setten, of omdat een programmeur simpelweg niet weet dat een bepaald stuk geheugen kritisch is in deze zin. Het zou een mooi en nuttig feature zijn van een multi HWD als hij dit soort fouten zou kunnen opsporen.

Als we ons bedenken dat een kritische sectie eigenlijk niets anders is dan een stuk geheugen dat tijdelijk ontoegankelijk (="guarded") verklaard kan worden, dan ligt het voor de hand om te onderzoeken of het bestaande memory map systeem van het MDS niet gebruikt zou kunnen worden om kritische secties te bewaken. Wat we dan moeten doen is zorgen dat als microprocessor A een semafoor set die sectie S bewaakt, er in de memorymaps van de andere geemuleerde processoren het bit "guarded" van het blok of de blokken waarin S ligt geset worden. Als dan een van die processoren ook in de sectie probeert te komen zal dit resulteren in een emulatiestop. Als processor A de sectie weer vrijgeeft, dienen de "guarded" bits weer gereset te worden.

Stel dat processor A vergeet de semafoor te setten voordat hij S binnengaat. Als dan een andere processor de semafoor wel set voor hij S binnengaat, zal dit tot gevolg hebben dat in de memorymap van A de sectie "guarded" wordt gemaakt, zodat de emulatie zal stoppen als A nogmaals een operatie uitvoert in S, waarmee dus ook zo'n fout is opgespoord.

Om een en ander te kunnen implementeren is het volgende nodig:

We moeten allereerst weten welke adressen semaforen bevatten. Bizonder geinteresseerd zijn we in de momenten waarop naar de semafooradressen wordt geschreven, want dat wil zeggen dat zo'n semafoor geset of gereset wordt. Op dat moment moet nl de emulatie gestopt worden en is de taak aan de debugprocessor om de betreffende geheugenblokken "guarded" te maken resp hun oude status terug te geven.

Het is natuurlijk mogelijk om het hele blok waarin een semafoor ligt "guarded" te maken. Als dan een emulatiestop optreedt moeten we bepalen of dit gebeurt t.g.v. acces tot een geheugen dat echt "guarded" is, of dat het hier een blok betreft waarin een of meerdere semaforen voorkomen. In het laatste geval moet nog worden gekeken of het hier wel of geen semafooradres betreft. In dat laatste geval moet de emulatie doorgestart worden, zoniet dan moet de "kritische sectie routine" in actie komen. Deze methode heeft 2 nadelen:

- ten eerste moet bij elke emulatiestop worden gekeken of het wel een echte is, of dat gestopt is omdat in het blok een semafoor ligt.
- ten tweede wordt erg vaak ten onrechte gestopt. Als er maar 1 semafoor in een blok ligt is de kans op een terechte stop 1 op 256 en zelfs 1 op 4096 bij een processor die 1 Mbyte kan adresseren.

Ik stel dan ook voor om een aparte memorymapping te maken voor semafooradressen. Deze zou kunnen bestaan uit een 4k x 1 ram, aangesloten op de 12 minst significante adreslijnen van de targetprocessor. De software instelling geschiedt volledig analoog aan die van de gewone memorymap. We kunnen het zo maken dat alleen als er geschreven wordt naar een adres waarvan de 12 minst significante adreslijnen overeen komen met die van een semafoor adres, de emulatie gestopt wordt.

Een stop op deze conditie kan ook een aparte status krijgen, zodat de debugprocessor meteen kan zien wat er aan de hand is als de emulatie gestopt wordt. De kans dat een stop niet terecht is is nu verminderd tot 15 op 16, (255 op 256 bij 1M adressen) terwijl het aantal stops met een factor 16 verminderd is. Ook weten we dat het hier alleen om een mogelijk semafooradres kan handelen.

De reden dat ik de minst significante adreslijnen kies is de volgende: Stel dat ik ga kijken naar de meest significante adreslijnen en mijn target systeem bevat maar de helft van het maximaal mogelijke geheugen. Dan weet ik al van te voren dat de helft van de blokken waarvan ik opgeef dat ze geen semaforen bevatten toch nooit zullen worden aangesproken. Als ik de minst significante adreslijnen beschouw is dit niet het geval.

Is de emulatie eenmaal gestopt dan moet de debugprocesor weer aan de weet zien te komen op welk adres er werd geschreven (dit kan het semafooradres zijn). Omdat we nu de lopende instruktie eerst af kunnen maken en geen geheugentransactie hoeven simuleren, zoals eventueel het geval zou zijn bij de software oplossing van eerder genoemd communicatieprobleem, is dit adres vrij eenvoudig te achterhalen. We zouden dit bv kunnen doen door het laatste woord van het real time trace memory uit te lezen. Omdat het traceboard optioneel is, is deze methode verwerpelijk. Wat we ook kunnen doen is latches maken op het RTB waarin elke geheugencyclus het adres wordt ingeklokt dat op de bus van de target processor staat, totdat gestopt wordt door de semafoormap. Nu moet de CMC deze latches kunnen uitlezen. Door daarna de targetprocessor de inhoud van het gevonden semafooradres te laten lezen (als het een semafooradres betreft) weet de debugprocessor welke data er naar het semafooradres zijn geschreven.

De debugprocessor zal voor het begin van de emulatie ook een aantal gegevens moeten krijgen nl:

- een lijst van semafooradressen. Hij kan daaruit dan zelf de semafoormap afleiden en bovendien bij een stop t.g.v. de semafoormap bepalen of het betreffende breakadres inderdaad een semafoor is.
- per semafoor welk bit, of combinatie van bits de semafoor functie vervult.
- per semafoor op welke adresruimte hij werkt.

De enige onvolkomenheid die we nu nog hebben is dat de emulatie ook zal stoppen als een adres aangesproken wordt dat niet tot een kritische sectie behoort, maar wel in hetzelfde blok ligt als een kritische sectie. Als dit als bezwaarlijk ervaren wordt, kunnen we dit door de debugprocessor laten detekteren. Deze kan de emulatie doorstarten als ten onrechte gestopt is. Zo'n test wordt dan wel elke keer uitgevoerd bij een emulatiestop t.g.v. toegang tot "guarded" memory. Ook in dit geval moeten we kunnen zien welk adres is aangesproken. Dat betekent dat de eerder genoemde adreslatches ook de adressen moet vasthouden bij deze emulatiestops.

Samengevat: Ten koste van redelijk wat extra hard-en software kan het MDS zo ingericht worden dat op een bevredigende manier kritische secties van target systemen bewaakt kunnen worden. Het is niet per se noodzakelijk dit te realiseren, door sommigen wordt zelfs in twijfel getrokken of er behoefte aan bestaat.

3.6 Interrupts

Zoals ook in hoofdstuk I is vermeld zal communicatie tussen microprocessoren vaak gepaard gaan met interrupts. Dit mechanisme wordt dan door de microprocessoren gebruikt om elkaar er op te attenderen dat er een bericht klaar staat of klaar gezet moet worden. Het zal duidelijk zijn dat communicatie die hier op berust niet binnen MDS getest kan worden als we niet de mogelijkheid bieden deze interrupts desgewenst te genereren. Algemeen kunnen we er van uit gaan dat in het prototype de interrupt ingangen van de microprocessoren, al dan niet via interrupt controllers, verbonden zijn met de uitgangen van I/O poorten, zodat een microprocessor een ander kan interrumperen door een I/O instruktie uit te voeren.

Real time deze interrupts genereren binnen MDS is ondoenlijk, omdat elke willekeurige output van elke I/O poort bij elke processor een bepaalde interrupt moet kunnen veroorzaken. Om dit allemaal te decoderen is geen haalbare zaak, dus moeten we onze toevlucht zoeken in simulatie, dus een software oplossing.

Allereerst moeten we detekteren of er een I/O poort wordt aangesproken die een interrupt zal veroorzaken. Dit is eenvoudig te realiseren door de betreffende outputpoort "in MDS" te definieren. Normaliter hebben we dan de mogelijkheid dat de informatie voor die poort naar het scherm gaat, of op diskette wordt gezet. Nu moeten we een derde mode introduceren, hier "interrupt mode" genaamd. Data die naar zo'n poort geschreven worden moeten dan door de debugprocessor gebruikt worden om een interrupt te genereren.

Een interrupt zouden we kunnen simuleren door eenvoudigweg bij de te interrumperen microprocessor een restart adres op te drukken. Dit heeft als ontoelaatbaar bezwaar dat zo'n restart adres ook wordt opgedrukt als de betreffende interrupt disabled of gemaskeerd is. Dus zouden we eerst moeten gaan kijken of de interrupt die we willen simuleren toegestaan is. Dit is niet bij elk type microprocessor mogelijk en daarom niet toepasbaar. Ook zouden we alle commando's bij kunnen houden die op de interrupts werken. Dit kunnen er vrij veel zijn, en bovendien kan het effect van zo'n instructie nog afhankelijk zijn van de inhoud van een register (bv SIM bij de 8085) en is daarom niet aantrekkelijk.

De enige mogelijkheid die ons dan nog rest is rechtstreeks op de interrupt ingangen van het processor IC de interrupts op te drukken. Dit vergt dus een aanpassing van de MAB's, omdat alleen daar de interrupt ingangen toegankelijk zijn voor de debugger. Er moeten in de MAB's I/O poorten komen die ingesteld kunnen worden door de CMC en waarvan de respectievelijke uitgangen verbonden zijn met de interrupt ingang(en) van de targetprocessor.

Als er door een microprocessor een I/O poort wordt aangesproken die ergens anders een interrupt moet genereren dan zal de emulatie stoppen, de debugprocessor zal in de betreffende MAB een I/O uitgang activeren (via de CMC) en als de emulatie dan doorgestart wordt moet deze output meteen enabled worden, zodat dan de betreffende targetprocessor geinterrumpeerd wordt.

De debugprocessor zal de volgende gegevens moeten krijgen:

- De outputpoorten die interrupts moeten genereren. Hieruit kan dan het stuk van de outputmaps afgeleid worden voor zover het betrekking heeft op 'interrupt' outputpoorten. Bovendien zal er in een lijst bijgehouden moeten worden welke poorten deze status hebben.
- Per poort moet bekend zijn welk bit of welke bits interrupts kunnen veroorzaken.
- Per bit moet aangegeven worden welk type interrupt gegenereerd moet worden. Dit nu is targetprocessor afhankelijk. Deze afhankelijkheid zit echter alleen in de debugprocessor omdat daar een vertaling gemaakt kan worden van de naam of het nummer van de interrupt naar het nummer van de overeenkomstige bit in de outputpoort in de MAB. Is dit nog bezwaarlijk dan zal de gebruiker zelf dit nummer op moeten geven en zal hij dus in zijn handleiding moeten gaan bladeren.
- Van elke interrupt moet vermeld worden welke HWD de te interrumperen microprocessor bevat.
- Indien nodig moet ook een restart adres of een bepaalde instructie worden opgegeven die de targetprocessor moet uitvoeren zodra hij de interrupt ziet.
- Eventueel kan nog gedacht worden aan het toekennen van prioriteiten voor het geval meerdere interrupts staan te wachten om afgehandeld te worden.

Met hetgeen tot nu toe beschreven is, is de debugprocessor in staat een CMC opdracht te geven om in de bijbehorende MAB een poort zo te sturen dat de targetprocessor een interruptrequest krijgt. Vaak zal de targetprocessor als hij een interrupt honoreert, middels een interrupt acknowledge cycle, vragen om een restart adres of een instructie die uitgevoerd moet worden. In zo'n geval moeten we zorgen dat de juiste code of codes op de databus opgedrukt worden.

Als de target microprocessors altijd tijdens een INTA-cycle maar 1 woord vragen, zouden we een I/O buffer kunnen maken, paralel aan de databus, waarin de CMC dit woord kan zetten, zodat het op het juiste moment op de databus van de targetprocessor kan komen te staan, door de uitgang te enablen met het signaal INTA. De CMC moet er voor zorgen dat hierin altijd het woord staat, dat behoort bij de hangende interrupt met de hoogste prioriteit. Als we toelaten dat er meerdere interrupts staan te wachten moet de CMC kunnen zien wanneer er een INTA-cycle is geweest om daarna (tijdens een emulatieonderbreking) een volgende restartcode in het buffer te zetten.

Als de target processor tijdens de INTA-cycle geen restartadres maar een instructie verwacht zouden we via genoemd buffer een "no operation" code (1 byte) kunnen opdrukken tijdens de INTA-cycle, om meteen daarna de emulatie te stoppen en de targetprocessor geforceerd de gewenste instructie te laten uitvoeren.

Het is ook eenvoudig mogelijk te maken om via het toetsenbord elke interrupt te genereren die men wenst, hetgeen ook in single processor systemen van voordeel kan zijn om interrupt routines te testen.

IV FOUTENDETEKTIE IN GEREALISEERDE SYSTEMEN.

4.1 Inleiding

In het vorige hoofdstuk hebben we ons bezig gehouden met niet- of ten dele voltooide systemen. Nu zullen we bekijken welke faciliteiten nodig, nuttig of handig zijn bij het opsporen en analyseren van fouten in systemen die (grotendeels) in de hardware gerealiseerd zijn. De in het vorige hoofdstuk besproken punten zijn mijns inziens noodzakelijk voor een multi-ICE. Voor de punten die hier ter sprake komen geldt meestal dat we het nut moeten afwegen tegen de kosten.

4.2 Logic state analyzer

Het meest nuttige instrument voor het opsporen van fouten in microprocessor omgevingen is op dit moment de logic analyzer. De functie van dit apparaat, in het bizonder van de logic state analyzer wordt in het MDS vervult door de combinatie van het "traceboard" en de trigger- en qualifier logica op het "real time board". Hieronder zullen enige uitbreidingen worden genoemd van dit samenstel die nuttig kunnen zijn in multiprocessor omgevingen.

- Gecombineerde triggermogelijkheid

Bij single ICE hebben we de mogelijkheid om te triggeren als triggerwoord 2 is gevonden nadat N keer triggerwoord 1 is gevonden of omgekeerd. Voor multi-ICE moeten we deze faciliteit uitbreiden zodanig dat debugger2 getriggerd wordt als debugger1 N keer aan een triggerconditie heeft voldaan, en via doorlussen zouden we zelfs een gecombineerde triggerfunctie kunnen maken voor 3 of 4 debuggers. Een en ander is nuttig als bij microprocessor 2, of in het deelsysteem dat via de userprobes van debugger2 bekeken wordt fouten optreden nadat eerst in deelsysteem 1 aan bepaalde condities is voldaan, of wanneer men vermoedt dat dit het geval is.

In beperkte mate hebben we die triggermogelijkheid nu al. Op de MAB's zitten nl uitgangen die actief worden als triggerwoord 1 resp triggerwoord 2 optreden. Door een van de userprobes van debugger2 met een van deze uitgangen te verbinden kunnen we debugger2 triggeren nadat een triggerwoord in systeem 2 is gevonden N keer na een triggerwoord in subsysteem 1. Veel machtiger zouden we zijn als op de MAB's niet deze twee uitgangen, maar een gecombineerde uitgang, en een triggeringang aanwezig waren. De uitgang moet dan actief worden als aan de totale triggerfunctie van de bijbehorende debugger is voldaan. De triggeringang moet, indien gebruikt, de functie overnemen van een van de twee triggerwoorden, of zelfs als derde triggervoorwaarde fungeren, eventueel gekoppeld aan een eigen matchcounter. Op deze manier is ook het eenvoudig doorlussen van triggersignalen via coax kabels met BNC connectors mogelijk .

- 30 kanaals analyzer

Met twee debuggers kunnen we beschikken over maximaal 32 userprobe lijnen. We zouden hiermee in principe een 32 kanaals logic state analyzer kunnen maken. Hiermee zouden we dan bv een 24 kanaals gemeenschappelijke adresbus of een 16 bits databus plus enkele andere signalen gelijktijdig kunnen bekijken. Dit is voor multiprocessor systemen veel belangrijker dan voor enkelvoudige systemen, omdat deze laatste vrijwel altijd opgebouwd zijn rond een enkel bussysteem, zodat we de meeste relevante signalen ook kunnen meten op de uit- en ingangen van de microprocessor en de userprobes dus beschikbaar houden voor andere signalen. Bij multi microprocessor systemen kunnen op de verschillende bussen na elkaar gegevens van verschillende bronnen aanwezig zijn. Deze kunnen we alleen goed bekijken door user probes aan zo'n bus aan te sluiten.

Per debugger hebben we bij de userprobe lijnen een kloksignaal nodig, dat wordt betrokken van 2 probe lijnen (1 per debugger) die in principe op hetzelfde signaal moeten worden aangesloten. Hierdoor houden we nog 30 data kanalen over.

Door van beide modules de triggeruitgangen te verbinden met de triggeringangen zoals die zijn voorgesteld bij het vorige punt, kunnen we triggeren op een 30 bits woord, mits er binnen elke debugger een and-functie mogelijk wordt gemaakt tussen de triggeringang en een van de triggerwoorden. Merk op dat de aparte triggeringang voor deze toepassing onontbeerlijk is !

Als de and-functie van triggeringang en triggerwoord van beide modules ook nog verbonden kan worden met een matchcounter hebben we een logic state analyzer van 30 kanalen die kan triggeren nadat N keer een bepaald triggerwoord is opgetreden.

Het koppelen van de qualifiers van twee ICE modules is moeilijker omdat er, afgezien van de "force trace lijn", naar buiten toe geen qualifier uit- of ingangen zijn. We kunnen hooguit aan beide groepen een of meer kanalen onttrekken en deze als qualifiers gebruiken. Nu moeten wel per se voor elke qualifierfunctie uit elke groep een lijn op het qualifier signaal worden aangesloten.

Het presenteren van de verschillende kanalen in de gewenste vorm op het scherm is voor de software een fluitje van een cent.

- sequentiele presentatie van de inhoud van twee trace memories

Een ander punt dat te maken heeft met presentatie op het scherm is het volgende: Stel dat we een programma willen bekijken dat loopt op een bepaalde microprocessor en vervolgens wordt overgenomen door een tweede en daarna eventueel nog door een derde, of weer door de eerste etc, dan is het handig als de presentatie op het scherm van de tracegegevens hiermee meeloopt. D.w.z dat de gebruiker het tracememory van ICE 1 kan uitlezen, maar vanaf het punt waarop het programma wordt overgenomen door processor 2 moeten de gegevens van het daarbij behorende tracememory worden gepresenteerd. Omwille van de eenvoud zullen we ons beperken tot situaties waarin een programma maar van twee processoren gebruik maakt. Zijn dit er meer dan moeten we dat programma in

meerdere fasen bekijken.

Bij een dergelijke vorm van presentatie zal de gebruiker tevoren op moeten geven tot welk punt (=triggerwoord) het programma op processor 1 loopt en vanaf welk punt op welke processor het wordt voortgezet. Dit laatste zal vaak een restartadres na een interrupt zijn. Het is dan de taak van de debugprocessor om te zorgen dat in de eerste debugger het ene woord als triggerwoord wordt ingesteld, waarbij de analyzer in pre-triggermode moet worden gezet. Bij de tweede debugger moet zijn triggerwoord ingesteld worden en moet de post triggering gekozen worden. Nadat de emulatie gestopt is zal de debugprocessor "real time trace memory" 1 uitlezen en de gevraagde regels op het scherm brengen. Bij elke regel moet gekeken worden of hij overeenkomt met het eerste triggerwoord, zo ja dan moet begonnen worden RTTM 2 uit te lezen en vanaf triggerwoord 2 moeten de gegevens op het scherm komen. Bovendien moet de gebruiker steeds kunnen zien welk trace memory hij aan het bekijken is.

- detekteren van "multiple acces"

Een fout die kan optreden bij multiprocessorsystemen is dat meerdere masters tegelijk toegang krijgen tot een gemeenschappelijke faciliteit zoals bv een bus, een geheugen module of een I/O apparaat. Een manier om de integriteit van een gemeenschappelijk onderdeel vast te stellen zou zijn het systeem te emuleren waarbij een emulatiestop optreedt bij multiple acces. We kunnen dit detekteren als we ons bedenken dat er in een systeem altijd signalen aanwezig zijn die aangeven of een bepaalde master toegang heeft tot een bepaalde faciliteit. Wanneer we bij een faciliteit userprobes aansluiten op deze signalen (voor elke master 1) en we kunnen de emulatie stoppen als meerdere lijnen gelijktijdig actief zijn, kunnen we zien of bij die faciliteit multiple acces optreedt.

Voor de realisatie is een 512x1 prom nodig: We sluiten 8 adreslijnen van het prom aan op 8 userprobe lijnen en de 9e op een software instelbaar signaal dat aangeeft of we een emulatiestop willen als meerdere lijnen hoog dan wel laag zijn. Met de CE- lijn geven we aan of we van dit feature wel of geen gebruik willen maken. Het prom moet zo geprogrammeerd worden dat de uitgang actief wordt als twee of meer ingangen gelijktijdig actief zijn. Deze uitgang kan direkt gebruikt worden voor het generen van een emulatiestop.

Er bestaan echter helaas geen 512x1 proms. We kunnen dan een 512x4 prom nemen en de resterende uitgangen voor andere dingen gebruiken of een 128x4 prom nemen en de uitgangen via selectielogica tot een uitgang samenvoegen.

- 9-bits pods

We kunnen op het ogenblik een van de userprobelijnen als clock en een of meerdere als qualifier gebruiken. Dit heeft als consequentie dat we deze lijnen niet meer als datakanalen kunnen gebruiken (tenzij we een qualifier signaal ook op het scherm willen zien). Dit nu is erg onhandig, omdat vrijwel alle bussen tegenwoordig een breedte hebben die een veelvoud van acht is. Het is daarom de moeite waard om te bekijken of het economisch haalbaar is om pods te fabriceren van 9 bits breed, zodat we 8 lijnen voor data kunnen gebruiken en de negende bij de ene pod van een debugger als clock, bij de andere pod als clockqualifier (de pods kunnen wel identiek zijn). Uiteraard moet in dit geval de IDB met twee lijnen uitgebreid worden en ook de qualifier logica er een ingang bijkrijgen.

- het testen van buffers

In multiprocessorsystemen komen noodzakelijkerwijs altijd veel buffers voor, vooral om te zorgen dat masters alleen op geschikte momenten toegang krijgen tot gemeenschappelijke faciliteiten. We onderscheiden adres- en data buffers, en eventueel commando buffers. De eerste twee zorgen dat adressen resp. data die uit de microprocessor komen op het juiste moment worden doorgegeven naar een ander deel van het systeem, dan wel dat van buiten komende data op het juiste moment worden doorgegeven aan een processor. De correcte werking van deze buffers kunnen we controleren door op de uitgangen de userprobes aan te sluiten (bij lees-operaties op de ingangen). De gegevens van de microprocessor en van de userprobes, dus van in- en uitgangen van de buffers kunnen we in nemen in het RTTM. Via een qualifier moeten we zorgen dat alleen gegevens worden ingenomen als de buffers open zijn. Door nu de informatie op in- en uitgangen van de buffers te vergelijken (deze zou hetzelfde moeten zijn) weten we of ze correct werken.

Ideaal zou zijn om via comparators dit vergelijken automatisch te laten gebeuren, zodat we een emulatiestop kunnen genereren bij ongelijkheid. Dit zou echter te veel IC's kosten en is daarom niet haalbaar. We kunnen ook het RTTM laten uitlezen door de debugprocessor en deze dan laten vergelijken of beide reeksen gegevens overal overeen komen, zoniet dan moet er een foutmelding komen met vermelding van de foutieve regels als dit er niet teveel zijn. We kunnen dan in een keer zien of er ergens in het RTTM verschil is tussen in- en uitgangen.

Het moet niet moeilijk zijn voor de software om maximaal 16 probelijnen te vergelijken met een overeenkomstig aantal in- of uitgangen van de microprocessor mits de userprobes in een gunstige volgorde zijn aangesloten.

4.3 Logic timing analyzer

Om timing diagrammen te verifieren en timing fouten op te sporen is een logic timing analyzer vrijwel onmisbaar. Er zijn dan ook plannen om het MDS te voorzien van dit gereedschap. Gedacht wordt hierbij aan vier kanalen. Ik ben, na analyse van een aantal oorzaken van mogelijke fouten en door eigen ervaring met een logic timing analyzer, van mening dat het wenselijk zou zijn om dit aantal uit te breiden tot 6, om zodoende alle relevante signalen in bijna alle gevallen gelijktijdig te kunnen bekijken.

Behalve de gebruikelijke features van zo'n analyzer zijn er nog enkele mogelijkheden die me nuttig lijken:

Als circuits te zwaar belast worden, waarbij de uitgangen te lang of altijd in een ongedefinieerde toestand blijven, kan dit aanleiding geven tot fouten. Met de normale digitale apparatuur kunnen we deze foutenoorzaak niet zien. Door het signaal van een probe toe te voeren aan twee comparators met verschillende referentie spanningen, bv V ol max en V oh min en de outputs van deze comparators via een exclusive or te combineren tot 1 signaal dat we op het timing analyzer scherm zichtbaar kunnen maken, kunnen we binnen de sample nauwkeurigheid zien hoelang een signaal in de tussentoestand verkeert.

Handig is ook als we in een timingdiagram twee cursors op willekeurige plaatsen kunnen zetten en waarbij dan onder, of boven in beeld direkt het tijdsverschil tussen de momenten die door de cursors worden aangegeven wordt weergegeven. Dit realiseren zal de taak worden van de software die bij de timing analyzer hoort.

Het spreekt voor zich dat de timing analyzer alle triggerfaciliteiten krijgt die nu beschikbaar zijn voor de state analyzer. Er wordt over gedacht om 1 timing kanaal te gebruiken ter synchronisatie met de kanalen van de state analyzer. Dit is mijns inziens niet nodig, omdat je bij timing analyse toch alleen maar geinteresseerd bent in een bepaald groepje signalen. Je wilt alleen vanaf een zeker moment meten en dit wordt gewaarborgd door het juiste triggerwoord in te stellen. Het is wel zinvol om de functie waarop getriggerd is in beeld zichtbaar te maken.

4.4 Statistieken

De opzet van een multiprocessorsysteem zal vaak gebaseerd zijn op intuitie, eventueel aangevuld met schattingen die resulteren uit min of meer summiere berekeningen. Nadat het ontworpen systeem is opgebouwd zal men graag een indruk hebben van de bezettingsgraden van bussen en I/O devices en van de mate waarin elke microprocessor van diverse faciliteiten gebruik maakt, om aan de hand van die metingen te kunnen bepalen of wellicht een herindeling noodzakelijk is. Een microprocessor kan dan bv meer of minder prive geheugen krijgen, de I/O apparatuur kan anders over het systeem verdeeld worden, misschien moet zelfs het hele bussysteem gewijzigd worden. Ook is het mogelijk dat bepaalde prioriteiten verlegd moeten worden.

Het is de taak van een MDS dat claimt geschikt te zijn voor het analyseren van multi microprocessor systemen om de mogelijkheden te bieden de gebruiker een inzicht te geven in bv:

- het totale busgebruik
- de bezetting van een bus door 1 microprocessor, en dat voor elke microprocessor
- de gemiddelde tijd dat een microrpcessor moet wachten op toegang tot een faciliteit
- het gebruik van een I/O device, totaal en door een microprocessor.
- het gebruik van een stuk geheugen, totaal of door 1 processor.

We kunnen in de metingen die nodig zijn om de gewenste informatie te krijgen drie soorten onderscheiden nl.

1) Meting van een aantal gebeurtenissen per tijdseenheid, bv het aantal aanvragen van een bepaald device door een processor.
2) Meting van het tijdspercentage dat aan een bepaalde conditie is voldaan, bv het meten van de bezettingsgraad van een bus.
3) Meting van een aantal gebeurtenissen als functie van een ander aantal gebeurtenissen, bv om te kijken hoeveel aanvragen voor een bepaald device van 1 bepaalde processor stammen.

Hiervoor zijn steeds twee "event counters" nodig; per ICE module kan er als optie een aanwezig zijn. We kunnen dus kiezen of we per debugger een tweede event counter aanbrengen. Dit maakt dan het "trace board" gecompliceerder en duurder. Door echter een drievoudige teller, de 8253, te gebruiken treden deze problemen waarschijnlijk niet op. Het alternatief is dat we van de gebruiker eisen dat hij bij minstens twee debuggers een trace board installeert. Deze oplossing heeft bovendien als bezwaar dat er een (externe) verbinding moet komen tussen de beide prints omdat we de tellers gelijktijdig moeten kunnen starten en stoppen.

Stoppen van de totale meting kan op vier manieren:

- met de hand
- als de teller die de meeste pulsen moet tellen is rondgeteld, dit kan echter te lang duren
- we zouden voornoemde teller terug kunnen laten tellen zodat eerder de time out optreedt.
- als er aan een triggervoorwaarde is voldaan

Bij metingen van de eerste soort zal de ene counter real time clock pulsen moeten tellen, de ander zal bepaalde gebeurtenissen moeten tellen. Dit is te realiseren door als triggerwoord de voorwaarde te kiezen waaraan de te meten gebeurtenis voldoet. Nadat de emulatie gestopt is moet de debugprocessor de beide tellers uitlezen en de inhoud presenteren als bv het aantal gebeurtenissen per seconde.

Bij metingen van de tweede soort moeten beide tellers real time clock pulsen tellen, maar de ene mag dit alleen doen als aan een qualifierfunctie is voldaan. Die qualifier zou kunnen zijn signaal "BUS BUSY". De eerste teller telt dan alleen real time clock pulsen als de bus in gebruik is. De software moet de standen van beide tellers op elkaar delen. We weten dan het percentage van de tijd dat de bus in gebruik is.

Bij de derde soort metingen moeten beide tellers het aantal keren dat een bepaald triggerwoord optreedt tellen. Voor de ene teller zou het triggerwoord zo gekozen kunnen worden dat de teller opgehoogd wordt elke keer als een bepaalde processor een device aanvraagt. De ander moet opgehoogd worden telkens als dat device aangevraagd wordt. Door beide waarden op elkaar te delen weten we het percentage van de aanvragen dat van die processor stamt.

Samengevat: Om statistische gegevens te verzamelen moeten we kunnen beschikken over twee eventcounters. We moeten beide exact gelijk kunnen starten en stoppen. De ene moet real time clock pulsen kunnen tellen of triggerpulsen zoals op het ogenblik al mogelijk is. De ander moet real time clock pulsen kunnen tellen als aan een qualifier is voldaan. Dit betekent dat een van de twee qualifiers of beide met deze eventcounter verbonden moeten worden. Ook moet deze counter, evenals de ander, triggerpulsen kunnen tellen. De debugprocessor moet beide tellers kunnen initialiseren en na een emulatiestop de inhouden op de gewenste manier kunnen presenteren op het scherm of een printer.

4.5 Geheugentests

De enige manier om de geheugens van een targetsysteem goed te testen is via testprogramma's. Deze programma's zijn zo universeel toepasbaar dat het zinvol is om ze standaard bij de debugger mee te leveren. De gebruiker hoeft dan maar op te geven welk geheugengebied hij getest wil hebben.

Voor de tests kan gebruik gemaakt worden van de software die de gebruikersprogramma's in het geheugen van het targetsysteem laadt en weer kan teruglezen. Het bezwaar dat hier aan kleeft is dat deze tests dan vrij langzaam worden. We zouden de tests ook kunnen laten sturen door de CMC, maar deze moet dan hele stukken target processor afhankelijke code ter beschikking krijgen. Een derde mogelijkheid is nog dat we de testprogramma's in een hogere programmeertaal, bv PASCAL schrijven. De gebruiker moet ze dan zelf compileren alvorens te gebruiken. Het object programma kan hij dan in een stukje systeem geheugen laden (dit kan emulatiegeheugen zijn) en van daaruit de tests uitvoeren. Testresultaten kunnen dan via een "in MDS" gedefinieerde I/O poort naar het scherm worden gestuurd.

4.6 Bus sluitingen

Het is wellicht nuttig om voor de start van een emulatie even de adres- en databus te controleren op sluitingen, hetzij onderling, hetzij naar een voedingsspanning. Dit is eenvoudig te controleren door enkele uitgekiende testpatronen naar listig gekozen geheugenplaatsen te schrijven. M.b.v. de trace faciliteit kan men kijken wat er werkelijk op de bussen komt te staan. De debugprocessor kan constateren of dit de gewenste bitpatronen zijn en zo niet een foutmelding genereren en het debugproces stoppen.

I.p.v. het RTTM is het wellicht mogelijk het "break diagnost memory" te gebruiken voor het opslaan van de teruggelezen testpatronen. Dit is dan gunstiger omdat dit stuk geheugen altijd aanwezig is. (Het "break diagnost memory" is een tussengeheugen op het "real time board" voor tijdelijke opslag van gegevens. Het is toegankelijk voor zowel CMC als target processor.) Het is waarschijnlijk voldoende om als testpatronen 1010.... en 0101.... te kiezen voor zowel adres- als data bus.

Dit feature heeft natuurlijk niet specifiek met multi-ICE te maken, maar is iets dat me uit de praktijk is voorgesteld om op te nemen in het MDS.

PHILIPS

Geraadpleegde litteratuur

1) iSBC single board computer
hardware reference manual, Intel

2) P.H.Enslow,Ed., Multiprocessors and parallel processing
New York: Wiley Interscience, 1974

3) J.L.Baer, "Multiprocessing systems"
IEEE Transactions on Computers, vol. C-25, no 12
dec 1976

4) A.J.Weissberger, "Analasis of Multiple-Microprocessor
System Architectures"
computer design, june 1977

5) G.Adams and T.Rolander, "Design Motivations for Multiple
Microcomputer Systems"
computer design, march 1978

6) A.D.Marathe and A.K.Chandra, "Hardware link directs mul-
ticomputer interactions in proces control"
computer design, feb. 1979

7) I.Smith, "Microprocessor system design"
microprocessors, vol 2, no 1, feb 1978

8) C.Kinnie and M.Maerz, "Dual-port ram hikes throughput in
input/output controller board"
electronics, 17 aug 1978

9) E.W.Dijkstra, "Solution of a problem in concurrent pro-
gramming control"
communications of the ACM, vol 8, no 9, sept 1965

10) B.H.Liebowitz, "Multiple processor minicomputer systems"
part 1 + 2,
computer design, oct.+nov 1978

11) C.Pick a.o., "Multimicroprocessor computer system"
microprocessors and microsystems, vol 3, no 4, may 1979

12) J.Braun and G.White, "Parallel processing with minicompu-
ters increases performance, availability"
electronics, 5 july 1979

MULTI-PROCESSING

NAME NAAM Han v.d. Broek | SUPERS. VERV. | SH. BL. | SH. BL. 3 —

PROPERTY OF EIGENDOM VAN N.V. PHILIPS' GLOEILAMPENFABRIEKEN-EINDHOVEN-NEDERLAND | CHECK CONTR. | DAT. | FORM. A4

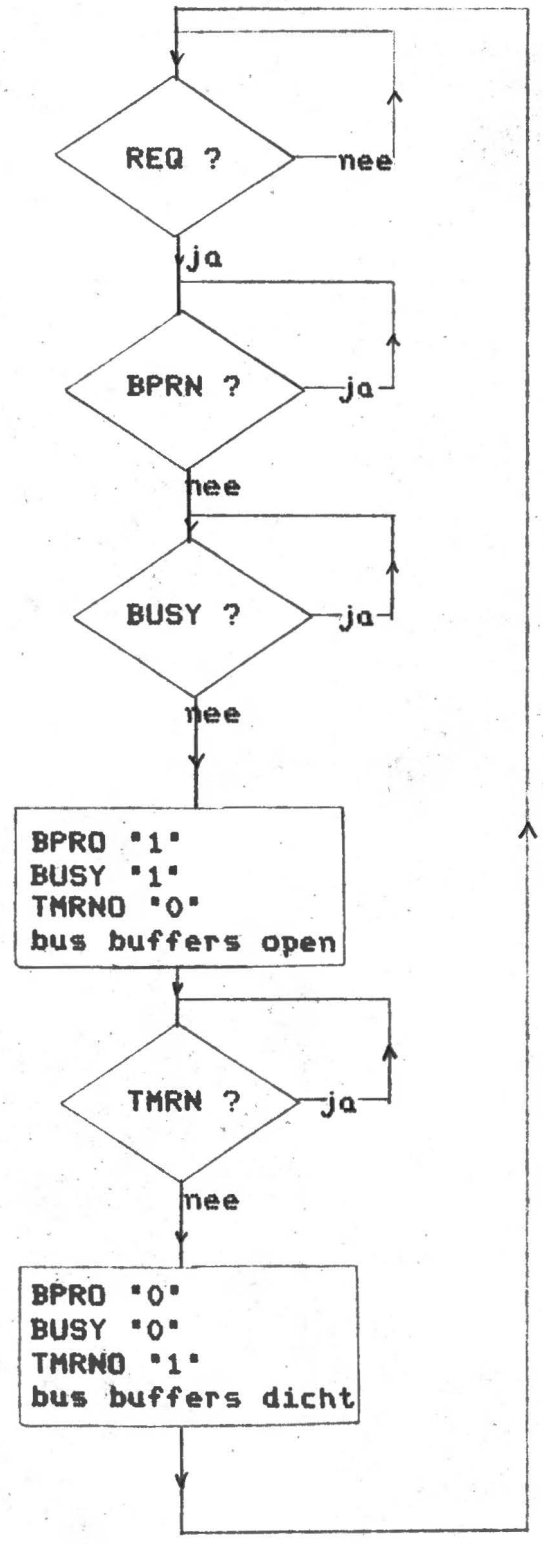
BIJLAGE I

Als we een verbinding willen maken tussen meerdere prive emulatiebussen en een gemeenschappelijke bus is een busarbiter nodig. Deze kan eenvoudig gerealiseerd worden (zie ook par. 1.5).

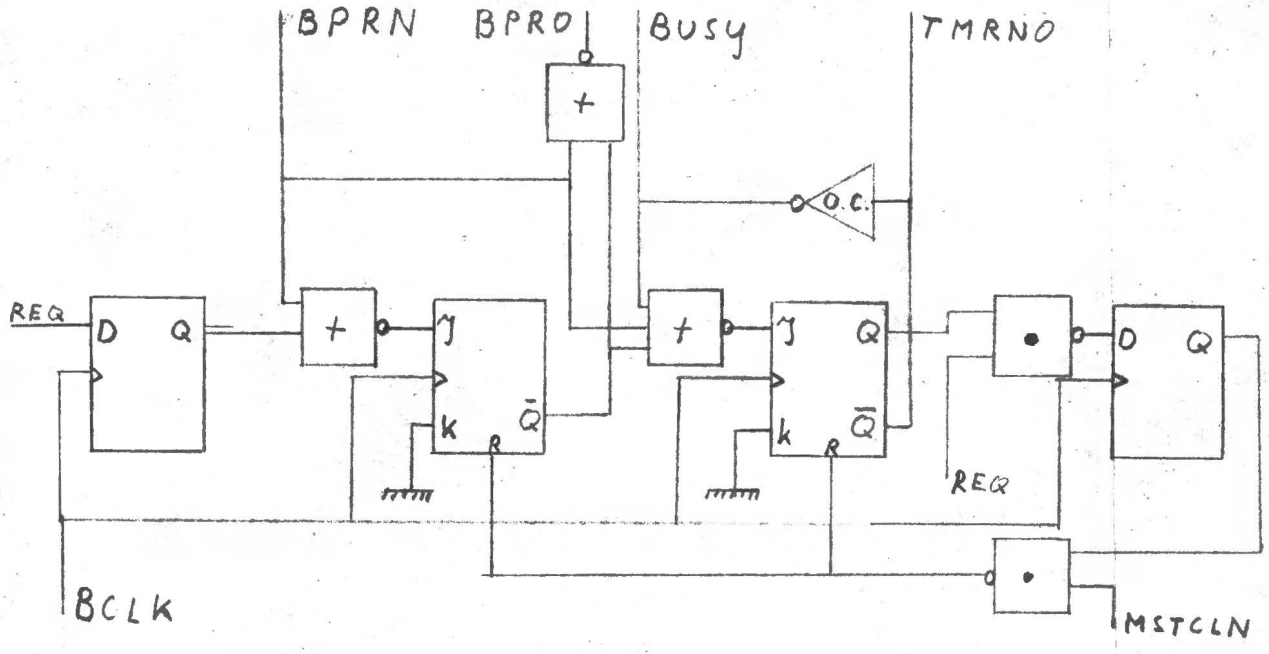
We werken met de volgende signalen:

REQ	request	dit is een active low signaal dat uit TSMN van het RTB wordt afgeleid als toegang tot het gemeenschappelijk emulatiegeheugen nodig is.
BPRN	bus priority in	een ingangssignaal. '0' geeft aan er zijn geen aanvragen van hogere prioriteit, '1' als deze er wel zijn.
BPRO	bus priority out	een uitgangssignaal, door te geven aan arbiters met lagere prioriteit, met dezelfde betekenis als BPRN.
BUSY	busy	gecombineerd in-uitgangssignaal. '1' geeft aan dat de huidige busmaster de bus in gebruik heeft, '0' als de bus is vrijgegeven.
TMRNO	TMRN out	uitgangssignaal dat aan het emulatiegeheugen wordt doorgegeven als timingssignaal voor de transfer.
MSTCLN	master clear	active low ingangssignaal; een algemene systeem reset.
BCLK	bus clock	een snelle klok die alle arbiters met elkaar synchroniseert.

Voor de arbitrage krijgen we het volgende toestandsdiagrams



De schakeling die dit realiseert wordt als volgt:



Deze schakeling volgt direkt uit het toestandsdiagram, alleen de extra input van BPRN op de derde flipflop is een listigheidje om te voorkomen dat er iets fout gaat als twee masters gelijktijdig BPRO hoog maken.

Zo'n schakeling moet er komen voor elke HWD, dus maximaal 4. Bovendien hebben we nog een snelle klok nodig, maar dit hoeft maar een eenvoudig oscillatortje te zijn. We zien dat we met vrij weinig IC's toekunnen (15 voor de totale arbitrage.