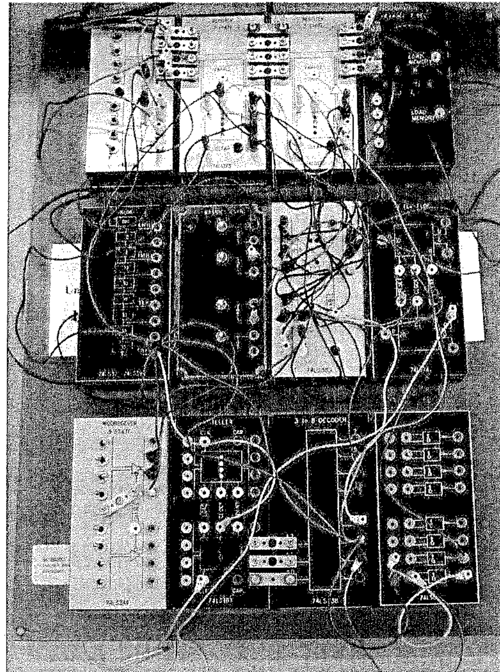
Hoofdstuk 12: Een busgeorganiseerde rekenmachine



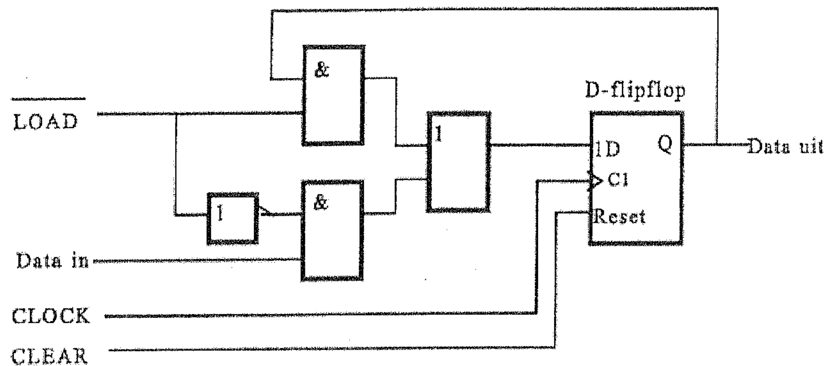
12.1 Inleiding en practicummateriaal

Bij het uitvoeren van een instructie door een computer worden gegevens (data) over de zogenaamde databus verzonden. Dit betekent dat informatie wordt uitgewisseld tussen verschillende registers waarbij meestal ook de ALU betrokken is. De databus bestaat uit een aantal lijnen waarop registers en de ALU zijn aangesloten. Verbindingen met de databus worden onderscheiden naar gelang het een ingang of een uitgang betreft. Daadwerkelijke verbinding met registeringangen komt tot stand door middel van een $\overline{\text{LOAD}}$ -signaal. Het moment waarop de informatie van de databus werkelijk in een register wordt opgeslagen, vindt plaats op het moment dat de systeemklok van niveau wisselt, dus tijdens een flank.
Aan de zijde van de uitgangen van de registers en van de ALU is een voorziening getroffen om te voorkomen dat meerdere uitgangen tegelijkertijd via de databus met elkaar verbonden zijn. Daarom is elk systeem dat een uitgang heeft op de databus, voorzien van een three-state (of 3-state) buffer. De functie daarvan is die van schakelaar waarmee de verbinding met de databus kan worden tot stand gebracht dan wel verbroken. De sturing van de three-state buffer geschiedt door middel van een signaal op een aparte ingang aangeduid met $\overline{\text{ENABLE}}$. Om de datatransfers van een instructie op het juiste moment en in de goede volgorde te laten plaats vinden is een sequencer nodig.
In dit hoofdstuk wordt een "accumulatormachine" in elkaar gezet met behulp van zogenaamde PIDAC-modules. In deze modules zijn geïntegreerde circuits (ic's) gemonteerd van de 7400 LSTTL serie. De schakelingen van de gebruikte ic's zijn beschreven in hoofdstuk 6 van het boek Van 0 en 1 tot processor. De kleur van de stekerbussen van de modules heeft de volgende betekenis:

- Geel en wit: ingangen
- Groen: uitgang
- Rood: in- of uitgang
- Zwart: 0 volt

12.2 Data-opslag in een register

Registers worden gebruikt voor opslag van tussenresultaten van berekeningen. Registers bestaan uit D-flipflops voorzien van een poortnetwerk. In figuur 1 is het schema van een één bits register weergeven.



Figuur 1: Eén bit register

Als de $\overline{\text{LOAD}}$ -ingang 1 is, staat op de uitgangen van de AND-poorten respectievelijk 'Data uit' en 0. De uitgang van de OR-poort heeft dus de waarde 'Data uit'. De flipflop zal op de eerstvolgende flank deze waarde overnemen. Dat is dus de onthoudfunctie. Als de $\overline{\text{LOAD}}$ -ingang 0 is, staat op de uitgangen van de AND-poorten respectievelijk 0 en 'Data in'. De flipflop zal op de eerstvolgende flank dus de waarde 'Data in' overnemen. Dat is de 'laad'-functie. Bij moderne digitale schakelingen wordt gebruik gemaakt van slechts één zogenaamde systeemklok. Deze is direct verbonden met de klokingangen van alle flipflops van de registers.

Het vier bit register 74LS173

Op het practicum wordt het register type 74LS173 gebruikt. Dit register is samengesteld uit vier flankgetriggerde D-flipflops en een viervoudige "2-line to 1-line data-selector" (zie ook boek: Van 0 en 1 tot processor fig. 6.6). Hieronder is de waarheidstabel van dit register weergegeven.

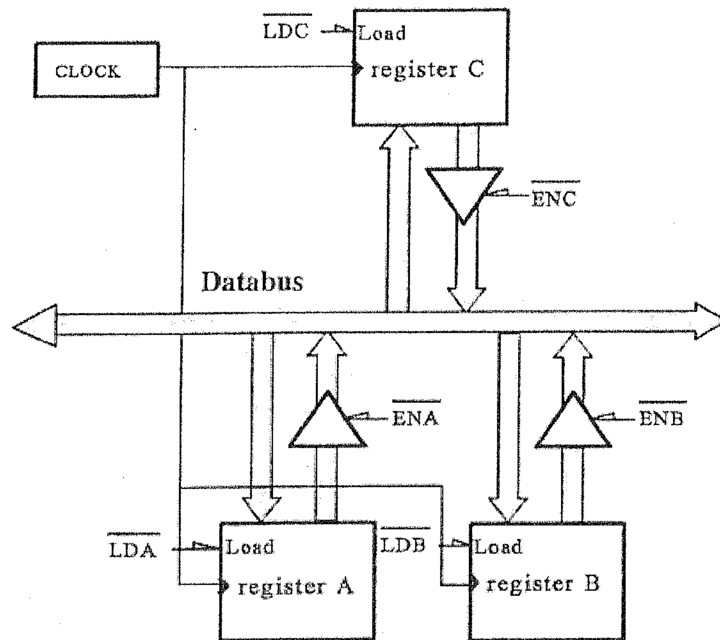
CLEAR	$\overline{\text{LOAD}}$	CLOCK	D_3	D_2	D_1	D_0	Q_3	Q_2	Q_1	Q_0
1	x	x	x	x	x	x	0	0	0	0
0	1	↑	x	x	x	x	q_3	q_2	q_1	q_0
0	0	↑	d_3	d_2	d_1	d_0	d_3	d_2	d_1	d_0

Tabel 1 waarheidstabel van het vier bit register 74LS173

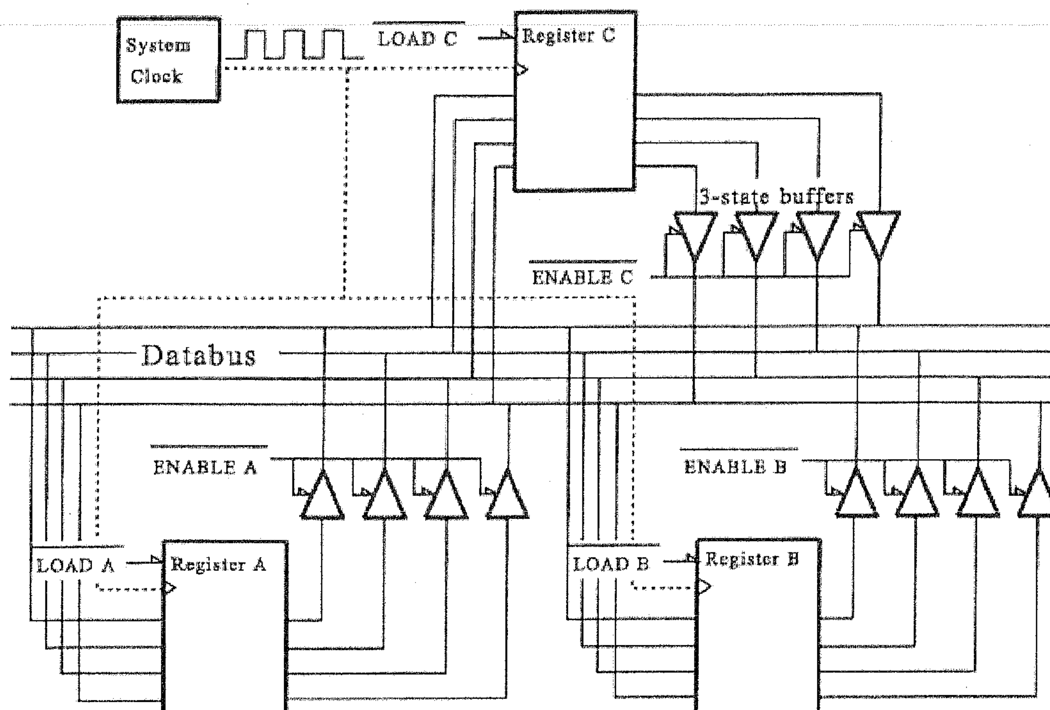
Uit de tabel blijkt dat het op nul zetten (CLEAR) plaats vindt als CLEAR 1 is, zonder dat er een klokflank nodig is. Het laden van een getal vindt plaats als $\overline{\text{LOAD}}$ is 0 en er een opgaande klokflank is. CLEAR is een asynchrone actie; laden is een synchrone actie.

12.3 Data-overdracht via een bus

In figuur 2 staat een schakeling weergegeven van drie registers, die via een databus met elkaar verbonden zijn. De bus waarlangs het datatransport plaats vindt, bestaat in de praktijk uit een aantal koperen printsporen. Bij veel recente processoren is dat aantal 64. Alle registers en andere computerbouwstenen die wat hun uitgang betreft, direct betrokken zijn bij data-uitwisseling, hebben voor elk spoor een three-state buffer. In de schakeling van figuur 3 is dit gedetailleerder weergegeven met een bus die bestaat uit vier sporen. Uit figuur 3 blijkt dat de buffers van één register door één enkele ingang, de $\overline{\text{ENABLE}}$-ingang worden bestuurd.



Figuur 2: Databus met drie registers



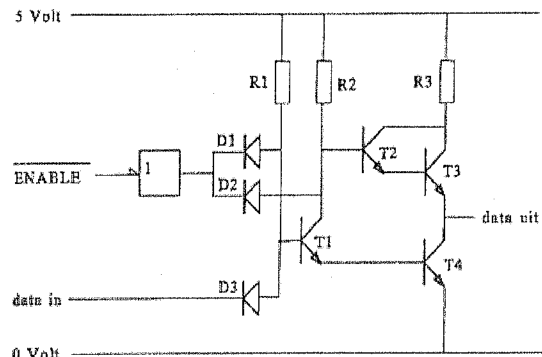
Figuur 3: 4 bit databus met drie registers

Als de $\overline{\text{ENABLE}}$ -ingang 1 is, is de uitgang zwevend. Praktisch gesproken betekent dit dat de elektrische weerstand tussen het register en de databus bijna oneindig groot is waardoor er geen contact is. Wanneer de $\overline{\text{ENABLE}}$ -ingang 0 wordt gemaakt is de buffer feitelijk een doorverbinding en is de waarde van de bufferuitgang gelijk aan die van de bufferingang.

Three-state buffer

Voor degenen die inzicht en ervaring hebben in het lezen van elektrische schakelingen, is in figuur 4 het elektronisch schema van de three-state buffer opgenomen. De drie mogelijkheden zijn als volgt:

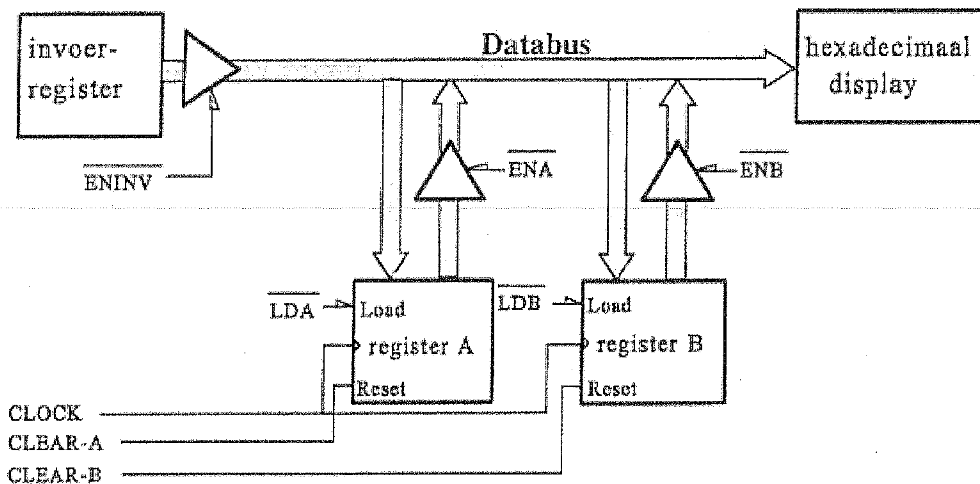
1. de uitgang kan 0 zijn. In dat geval geleidt transistor T4 en geleidt transistor T3 niet,
2. de uitgang kan 1 zijn. In dat geval geleidt transistor T4 niet en transistor T3 wel.
3. de uitgang is zwevend. Beide transistoren geleiden niet.



Figuur 4: Three-state buffer

Dataoverdracht tussen drie registers via een bus

In figuur 5 zijn opnieuw drie registers (register A, register B en een in voerregister) en een databus getekend. Verder is in de schakeling een hexadecimaal display opgenomen.



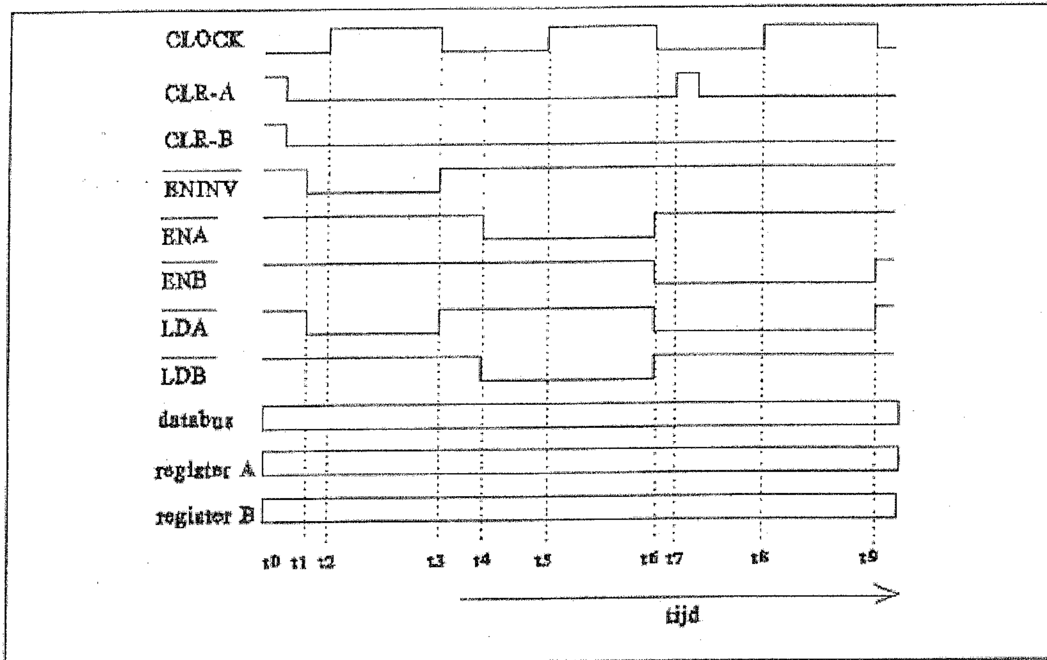
figuur 5: Data-transport via bus

Met deze schakeling kunnen de volgende datatransfers plaatsvinden:

- MOVE A, INV   Kopieer de inhoud van het invoerregister naar het A-register,
- MOVE B, INV   B ← INV,
- MOVE B, A   B ← A,
- MOVE A, B   A ← B.

In figuur 6 is weergegeven op welk moment besturingssignalen actief moeten zijn om

een datatransfer uit te voeren. Op tijdstip t_0 zijn alle $\overline{\text{LOAD}}$- en $\overline{\text{ENABLE}}$-ingangen 1, zodat alle registeruitgangen zwevend zijn. Tevens zijn de CLEAR-ingangen van de beide registers 1 waardoor de register-flipflops gereset zijn. Op tijdstip t_1 wordt $\overline{\text{ENINV}}$ 0. De inhoud van het invoerregister verschijnt dan op de databus. De waarde ervan is op het display af te lezen. Tegelijkertijd (dus op t_1) wordt de $\overline{\text{LOAD}}$–ingang ($\overline{\text{LDA}}$) van register A 0. De eerstvolgende positieve klokflank verschijnt op tijdstip t_2. Op dit moment wordt register A geladen. Op t_3 wordt de bus weer vrijgegeven voor een volgende transfer doordat $\overline{\text{ENINV}}$ en $\overline{\text{LDA}}$ beide 1 worden.



Figuur 6: Tijdvolgorde diagram datatransfers

Opdracht 1: Datatransfers
Beschrijf hieronder wat er gebeurt op de tijdstippen t_4 t/m t_9 van figuur 6.

- t_4:
- t_5:
- t_6:
- t_7:
- t_8:
- t_9:

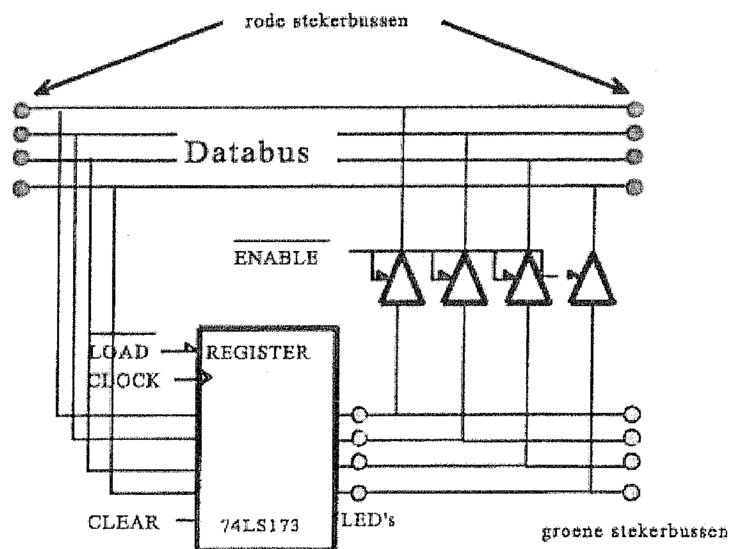
12.4 Practicum een busgeorganiseerde rekenmachine

In de volgende experimenten wordt stapsgewijs een busgeorganiseerde rekenmachine gebouwd waarmee operaties op twee vier bit getallen kunnen worden uitgevoerd.

- *In experiment 1 wordt de werking van three-state buffers bestudeerd.*
- *In experiment 2 wordt de werking van een module met register en three-state buffers uitgezocht.*
- *In experiment 3 wordt een schakeling gebouwd waarmee vier-bits data tussen registers via een bus wordt getransporteerd.*
- *In experiment 4 wordt een ALU aan de schakeling toegevoegd. Met de schakeling die dan ontstaat, kan de data bewerkt worden door achtereenvolgens de juiste datatransfers handmatig uit te voeren.*
- *In experiment 5 wordt een tweede invoerregister toegevoegd. Het is dan mogelijk langs twee verschillende wegen data in te voeren en met de ALU te bewerken.*
- *In experiment 6 wordt een sequencer gebouwd. Deze schakeling is nodig om iedere datatransfer op het juiste moment te laten plaatsvinden.*
- *In experiment 7 wordt een sequencer aan de schakeling toegevoegd. Nu is het mogelijk de datatransfers automatisch te laten uitvoeren.*

Experiment 1: Module met three-state invoerregister
Gebruik de bovenste helft van een module met opschrift "WOORDGEVER 3 STATE". Deze module bevat twee series van vier three-state buffers. De uitgangen van deze buffers kunnen in de zwevende toestand gebracht worden door op de gemeenschappelijke $\overline{\text{ENABLE}}$ -ingang een 1 te zetten. Sluit op deze module een HEXADECIMAAL DISPLAY aan. Maak de $\overline{\text{ENABLE}}$ -ingang achtereenvolgens 0 en 1 en verklaar de waarde op het display.

Experiment 2: Module met register en bus
De PIDAC-module met het opschrift REGISTER 3 STATE bevat behalve het register 74LS173, vier three-state buffers en een vier-bits databus. In figuur 7 zijn deze aangeduid als rode stekerbussen. Sluit op de linker rode stekerbussen een 3-state woordgever aan en op de rechter een display. Sluit op de



Figuur 7: PIDAC-module: "REGISTER 3 STATE"

CLOCK-ingang een WAARDEGEVER aan.

- Clear het register.
- Kopieer een getal van het invoerregister naar de 74LS173.
- "Disable" ofwel maak de $\overline{ENABLE}$ -ingang 1 van het invoerregister
- "Enable" ofwel maak de $\overline{ENABLE}$ -ingang 0 van de buffers tussen registeruitgangen en databus. Welke waarde geeft het display weer?
- Laad de waarde 3 in het register door de $\overline{LOAD}$ -ingang 0 te maken en daarna een opgaande klokflank te geven. Zet op het invoerregister de waarde 6. "Enable" beide $\overline{ENABLE}$ -ingangen. Welke waarde geeft het display weer? Verklaar je antwoord.

Dataconflict
Er mag dus hoogstens één databron op de bus beschikbaar/actief zijn!
Bij digitale circuits van het type LSTTL neemt een uitgang die de waarde 0 heeft, veel meer stroom op dan een ingang die de waarde 1 heeft kan leveren. Dus bij een kortsluiting "winnen" nullen het van enen. Bij ic's van het type CMOS is dat niet zo.

Experiment 3: Datatransport via bus

Bouw een schakeling waarmee 4-bit data uitgewisseld kunnen worden tussen een invoerregister en twee registers via een bus volgens figuur 5.
Gebruik als invoerregister een three-state woordgever (geel deksel) en als klok een waardegever.

- Laad een getal in register A en "disable" daarna de woordgever.
- Kopieer dit getal van register A naar register B via de databus.
- Clear register A en kopieer de waarde van B terug naar A.

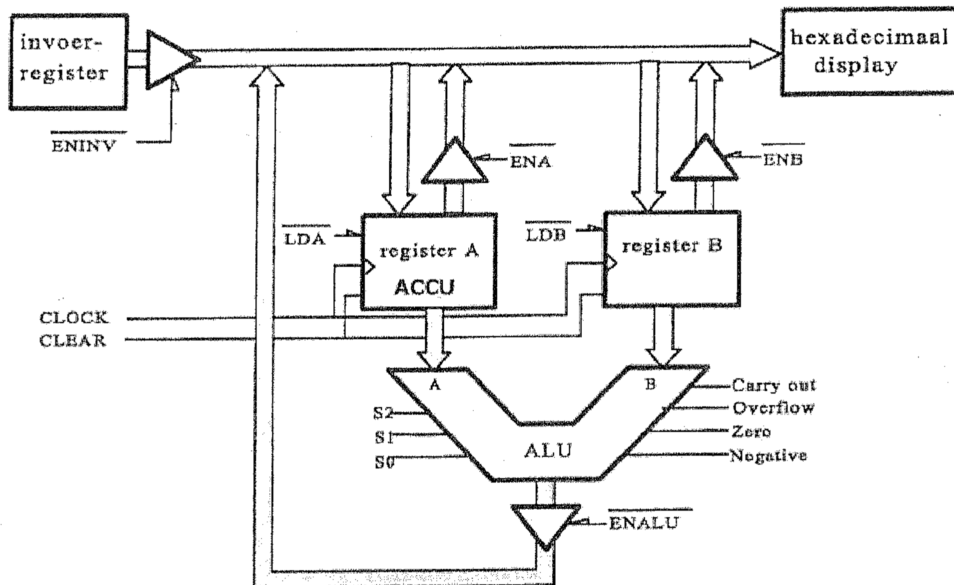
Experiment 4: Datatransfers via ALU naar registers

S_2	S_1	S_0	operatie	
0	0	0	$\overline{B}$	complement van B
0	0	1	B	B wordt doorgegeven
0	1	0	A - B	rekenkundige -
0	1	1	A plus B	rekenkundige +
1	0	0	$A \oplus B$	bitwise XOR
1	0	1	A + B	bitwise OR
1	1	0	A . B	bitwise AND
1	1	1	1111	-1

Tabel 5: Functietabel ALU

Modificeer de schakeling op de volgende wijze (zie figuur 8):

- Voeg aan de schakeling een ALU toe.
- Verbind de uitgangen van de registers vóór de buffers (groene stekerbussen) met de ALU



Figuur 8: Een busgeorganiseerde rekenmachine

Voer de volgende instructies uit:

- LDA # Load Accumulator; ACCU ← invoerregister.
- SUB # ACCU ← ACCU – invoerregister

Het uitvoeren van deze laatste instructie gaat in twee fasen :

fase 1: (hulp)register B ← invoerregister

fase 2: register A ← register A - register B.

Accumulatormachine
Een van de allereerste processoren maakte gebruik van slechts één register. In dit register werden alle tussenresultaten van ALU-bewerkingen opgeslagen. Een gebruikelijke naam voor dit register is ACCU. Dit is een afkorting van accumulator of verzamelregister.

Experiment 5: Een bus georganiseerde accumulatormachine
Voeg aan de vorige schakeling een tweede invoerregister toe. Voer de volgende instructie uit: SUB in1, in2 # ACCU ← invoerregister 1 - invoerregister 2.

Om deze instructie uit te voeren zijn 3 datatransfers nodig. Ofwel voor het executeren van deze instructie wordt een zogenaamd microprogramma bestaande uit micro-instructiestappen uitgevoerd. Schrijf dat programma door in tabel 2 de vereiste logische niveaus bij iedere datatransfer in te vullen.

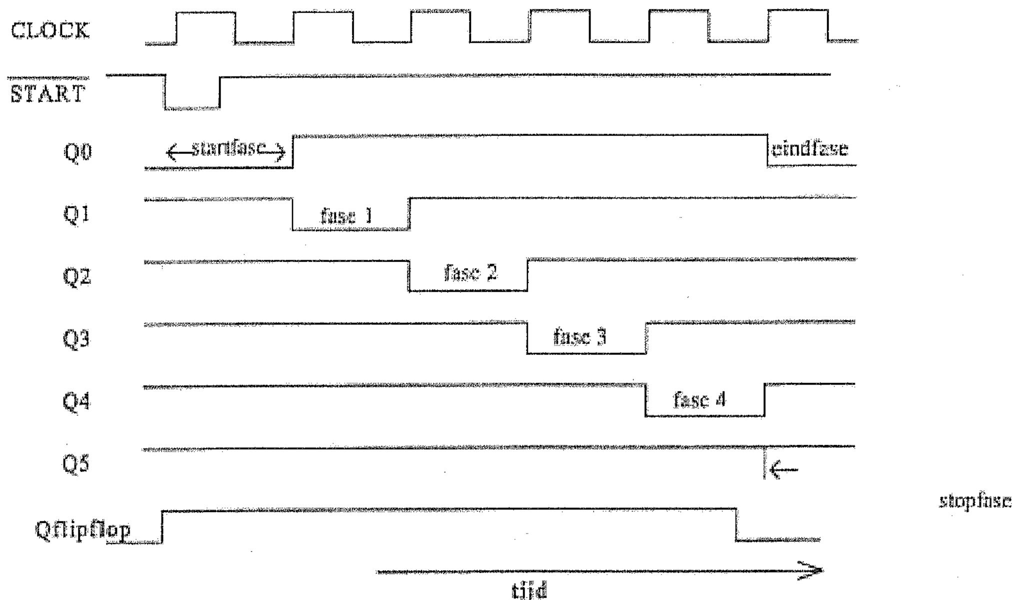
Houd de schakeling intact voor verdere experimenten!

fase / opdracht-ingang	bij aan-vang	fase 1	fase 2	fase 3	na afloop
$\overline{ENINV1}$	1	....	....	....	1
$\overline{ENINV2}$	1	....	....	....	1
$\overline{ENA}$	1	1	....	....	1
$\overline{ENB}$	1	1	....	....	1
$\overline{ENALU}$	1	1	....	....	1
$\overline{LDA}$	1	....	....	....	1
$\overline{LDB}$	1	....	....	....	1
S_2-ALU	x	x	....	....	x
S_1-ALU	x	x	....	....	x
S_0-ALU	x	x	....	....	x

Tabel 2: Logische niveaus van de stuuringangen voor iedere fase.

12.5 Sequencer

Om de achtereenvolgende fasen van een bewerking volledig automatisch te laten verlopen, wordt een sequencer gebruikt. In figuur 12 is weergegeven welke signalen een Sequencer moet genereren om bijvoorbeeld een tijdsafhankelijk proces, dat in vier

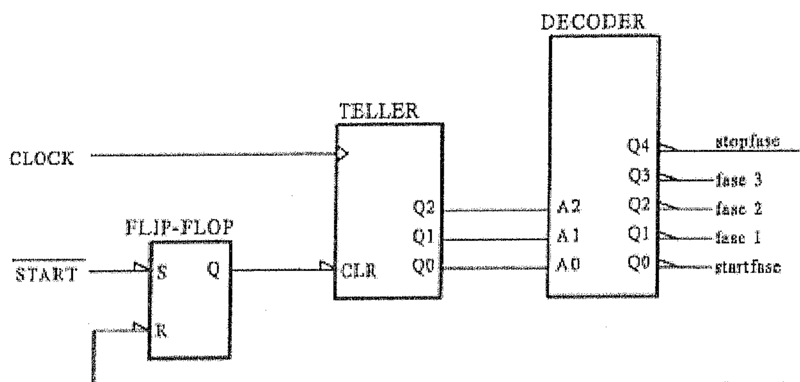


Figuur 12: Timing diagram van de sequencer

fases verloopt, te kunnen executeren. De cyclus wordt door een extern $\overline{\mathrm{START}}$ - signaal in combinatie met de eerstvolgende positieve klokflank gestart. Door een door de schakeling zelf gegenereerd $\overline{\mathrm{STOP}}$ -signaal wordt de cyclus beëindigd. Na de start begint iedere fase op een opgaande flank van een klokpuls. Elke actie wordt bestuurd door de klok en is dus synchroon.

In figuur 13 is de sequencer geïmplementeerd door middel van een vier-bits synchrone teller, een decoder en een Set-Reset-latch. Als teller gebruiken we het ic 74LS161. Als de COUNT-ingang 1 is wordt de tellerstand bij elke opgaande flank met 1 verhoogd. Als decoder gebruiken we een 3 naar 8 decoder type 74LS138. Deze decoder heeft "laag actieve" uitgangen. Afhankelijk van de code op de ingangen is één uitgang 0 en zijn de andere uitgangen 1. Voor de Set-Reset latch is het handig om de $\overline{\mathrm{SET}}$ en $\overline{\mathrm{RESET}}$ -ingangen van een D-flipflop type 74LS74 te gebruiken. Deze is opgebouwd uit NAND-gates (zie boek figuur 5.35)

Experiment 6: Bouw en test een sequencer met een start-, een stop- en drie signaalfases



Figuur 13: Sequencer

Experiment 7: Geautomatiseerde rekenmachine
Voeg aan de schakeling van experiment 5 een sequencer toe met een start-, een stop en drie signaalfases.
Sluit de uitgangen van de sequencer aan op de juiste $\overline{\text{LOAD}}$ - en $\overline{\text{ENABLE}}$ -ingangen.
Start de executie van dit experiment door één druk op een knop. Welke (druk)knop wordt hier bedoeld?
Eén van de opdrachtingangen, $\overline{\text{LDA}}$, moet in twee fases 0 worden. Ga na dat een AND-poort nodig is om deze ingang bij de juiste micro-instructiestappen te activeren.
Zorg ervoor dat steeds op de neergaande klokflank de volgende fase ingaat en dat op de opgaande flank steeds een register wordt geladen. De set-up time is hierdoor lang genoeg.

12.6 Begrippenlijst

Accu, accumulator. Het register waarin de tussenresultaten van ALU-bewerkingen worden opgeslagen (verzamelregister) bij een zgn. accumulatormachine.

Databus. Een aantal parallel lopende lijnen waarover het datatransport in een computer plaatsvindt. Naast datalijnen bevat een bus ook controlelijnen en adreslijnen. Een voorbeeld van een controlelijn is de lijn waarover het uitgangssignaal van de systeemklok wordt getransporteerd.

Datatransfer. De overdracht van data van het ene register naar het andere (al of niet via een bus en/of ALU).

$\overline{\text{ENABLE}}$ **-ingang.** Een ingang waarmee de waarde op uitgang van een three-state buffer wordt doorgelaten of geblokkeerd.

$\overline{\text{LOAD}}$ **-ingang.** Een ingang van een register waarmee (nieuwe) gegevens kunnen worden geladen. Bij de meeste registers is voor het laden van data ook nog een klokflank nodig.

Micro-instructie. Elementaire instructie zoals ADD of MOVE. Deze bestaat uit meerdere datatransfers.

Micro-instructiestap. Eén van de fasen waarin een micro-instructie geëxecuteerd wordt.

Sequencer. Schakeling die de timing genereert voor het uitvoeren van een sequentie van gebeurtenissen.

Three-state buffer. Een schakeling waarvan de uitgang of zwevend is of gelijk is aan de inputwaarde 0 of 1.

Chapter 13: The Memory Hierarchy

13.1 Introduction

Memory

The main memory of a computer system consists of Dynamic RAM cells. Each cell has only one transistor (see page 102). The time to read or write a DRAM cell degreases from 375 ns in 1980 to 40 ns in 2010. This is a factor 9 in 30 years. The access time for another memory type, Static RAM, a type with six transistors for each cell, degreases from 300 ns to 1 ns, a factor 300 in the same period. For the hard disk the seek time reduces from 90 ms to 3 ms, an improvement in the seek time of a factor 30 over this period. Hence it follows that different storage technologies have different performance tradeoffs. Unfortunately the best performance has the highest price.

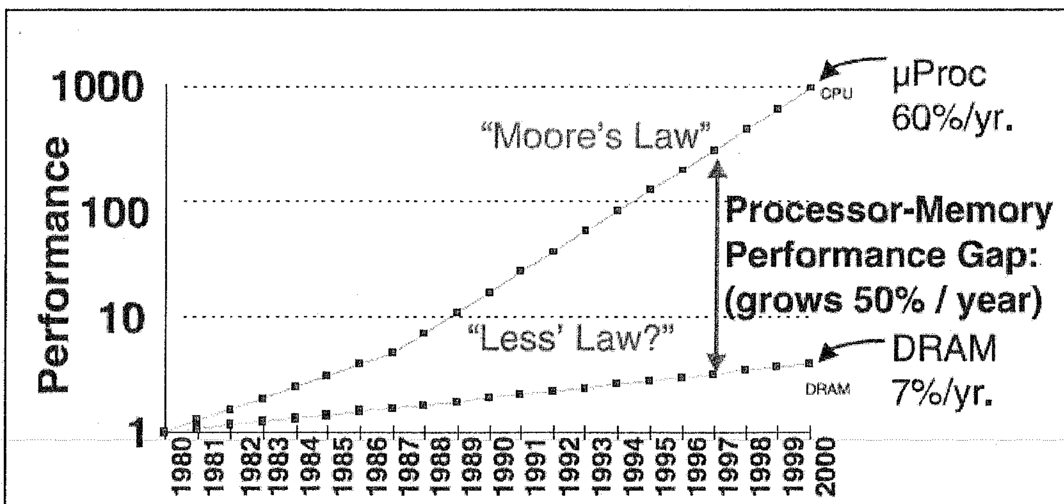


Figure 13.1: Processor-Memory Performance Gap

CPU

The CPU cycle time degreases from 1000 ns in 1980 to 0.4 ns in 2010, an improvement of a factor 2500. If we execute a Load Word-instruction with a Single Cycle Machine and the data comes from main memory (DRAM), the CPU waits 40 ns (100 cycles) for data. Amdahl's law tells us that further improvement of CPU-performance has no sense if this problem was not tackled.

Caches in the Memory Hierarchy

Caches were introduced in order to bridge the speed gap between processor and memory and are an essential element of today's high-performance microprocessors. The process of using a cache is known as caching.

Memory hierarchy

Figure 13.2 shows the memory hierarchy. The upper level is the register block, the edge-triggered registers are accessible within 0.1 ns. The size is 128 bytes (32 register * 4 bytes). The next level is the cache. De SRAM flipflops are accessible within 1 ns.

The Load Word and Store Word instructions transport 4 bytes between these levels.
From cache to the trid level ...

Memory hierarchy

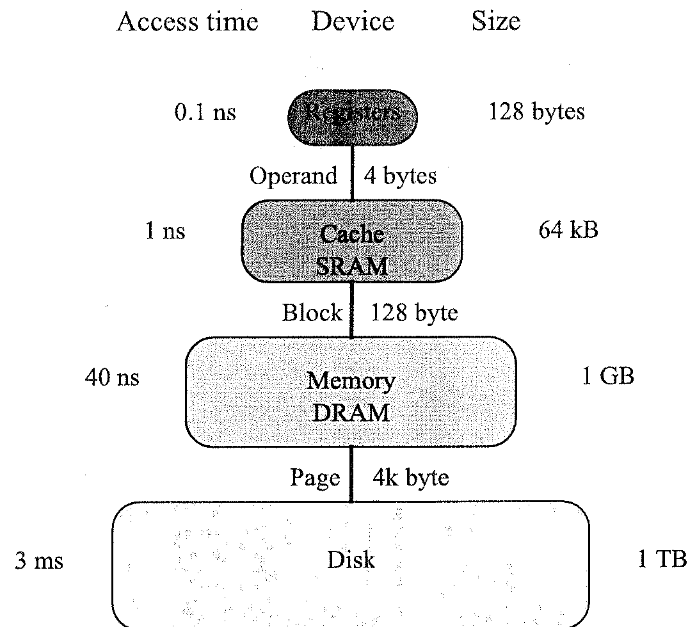


Figure 13.2

13.2 Caching

A cache is a fast and small SRAM memory which exploits *locality*. There are two types of locality:

- Temporal locality: a referenced item tends to be referenced soon again (e.g., instructions: loop structures)
- Spatial locality: items close to a referenced item tend to be referenced soon (data: arrays, instructions: instruction is most likely executed after instruction)

There are several types of caches which are being applied in modern processors. For example, Translation Look-aside Buffers (TLB's) are caches for storing recent virtual to physical address translations. These translations can be quite expensive in terms of performance as they often need multiple memory accesses to read the page-table. Luckily, TLB's are able to obtain hit ratios of over the 90% which prevents that virtual to physical address translation becomes a performance bottleneck.
Another well-known class of caches are the instruction and data caches. To this class belong the caches that only store instructions or data and those that are unified and store both instructions and data in a single cache. For the sake of discussion, we will focus on this class of data/instruction caches in the remainder of this section. The presented implementation methods are, however, also applicable to other types of caches.
Roughly speaking, there are three ways to implement the mapping of data located in the main memory to a location in the (much smaller) cache: direct mapped, fully associative and set-associative. In all implementations, the data granularity that is cached (and thus is mapped into the cache as a single entity) is called a *cache block*. Typically, cache blocks have a size of 16 to 256 bytes.

13.2.1 Direct mapped cache

In direct mapped caches, there is a one-to-one mapping of addresses in the main memory to cache locations using a modulo function. To illustrate this, let's assume the example cache in Figure 13.3. This cache has 8 entries which each can hold a cache block and the cache block size is 16 bytes. In a direct mapped implementation of our example cache, the first 16-byte block in main memory maps onto the first entry in the cache, the second block in main memory to the second cache entry, and so on. This scheme implies that main memory block 8 again maps to the first entry in the cache.

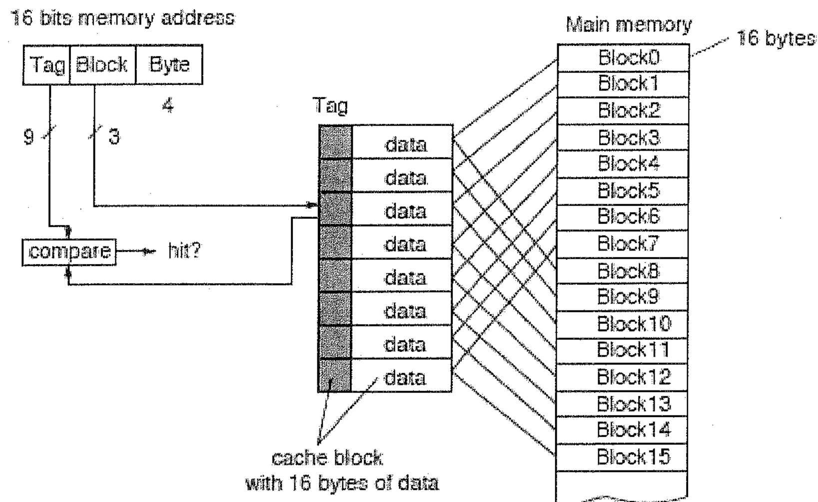


Figure 13.3: Direct mapped cache

Since multiple main memory blocks can be mapped to the same cache entry, the processor must be able to determine whether a data block in the cache is the data block that is actually needed. This is done by storing a *tag* together with a cache block. This tag constitutes the most-significant address-bits of the memory location the cache block in question stores. Consider figure 13.3 again. Here, we assume 16-bits addresses. Accessing the direct mapped cache with these 16-bits addresses is done as follows. The 4 least-significant bits are used to address the right byte in the 16-byte cache blocks. The next 3 bits (block index) determine in which cache entry the processor should look for the required data. So, these are the mapping bits. Finally, the remaining 9 most-significant bits are used for the tag. These tag bits are compared with the tag bits stored in the cache at the selected cache entry (selected by the 3 mapping bits of the block index). If the tags are identical, then there is a cache hit, implying that the data in the cache block is the data actually needed by the processor.
Direct mapped caches are, because of the straightforward mapping, relatively simple to build and allow for high-speed access. However, because this mapping is also very rigid, direct mapped caches may yield poor hit ratio. If you are accessing an array by element numbers 0, 8, 16, etc., for example, then every array access maps to the same cache location, thereby removing the data from the previous access. By making a direct cache very large, these effects can be reduced. For this reason, direct mapped caches are often used as 2 -level caches (which are often placed off-chip and can therefore be large) in processors.

Exercise 13.1: Cache size
Properties:

- A byte-addressable machine with 18-bit addresses for DRAM
- Direct-mapped cache
- Each block holds 16 bytes
- The block-index is 4 bits

Questions:

- How many blocks does the cache hold? ......
- How many bits are used for the tag? ....
- How many bits of storage are required for one *cache line* (a cache line is cache block plus tag and valid bit)? ....
- How many bits of storage are required to build the whole cache? ......
- Which part of the DRAM-memory can be loaded into the cache? ....

13.2.2 Fully associative cache

A fully associative mapping is more or less the opposite of a direct mapping. It allows for placing a cache block anywhere in the cache. According to some replacement policy, the cache determines a cache entry in which it stores a cache block. Consider Figure 13.4 to illustrate how such a cache is accessed. Again, we assume 16-bits addresses, a cache with 8 entries and cache blocks of 16 bytes. The latter means that the 4 least-significant bits of an address are again used to address the right byte in a cache block. There is no fixed mapping, so there are no mapping bits. Instead, because a required cache block can reside anywhere in the cache, all cache entries need to be searched. This means that the remaining 12 bits of an address are tag-bits and these tag-bits are compared in parallel with all tags stored in the cache. If there is a tag match, then there is a cache hit. In that case, the data from the cache block with the matching tag can be used by the processor.

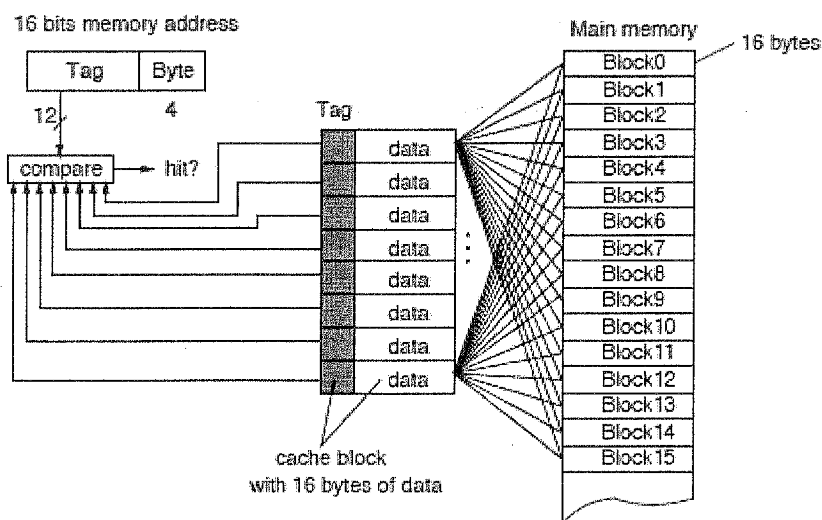


Figure 13.4: Fully associative cache

Clearly, fully associative caches have a flexible mapping which minimizes the number of cache-entry conflicts (i.e., they may yield high hit ratio). However, a price must be

paid since a fully associative implementation is expensive (e.g., it must be capable of doing a large number of comparisons in parallel). For this reason, appliances of fully associative caches never use large cache sizes. Examples of these appliances are TLB's and branch history tables (of which the explanation is beyond the scope of this syllabus).

13.2.3 Set-associative cache

A set-associative mapping can be seen as a combination of a direct mapping and a fully associative mapping. In a set-associative cache, the cache entries are subdivided into cache sets. Similar to a direct mapping, there is a fixed mapping of memory blocks to a set in the cache. But inside a cache set, which can hold several cache blocks, a memory block is mapped in a fully associative manner. To illustrate this, consider figure 13.5. It depicts a 2-way set-associative cache with 16-byte blocks. The "2-way" means that each set contains 2 cache blocks. This means that the 8-entry cache is subdivided into 4 sets. Memory block 0 maps to the first set, memory block 1 to the second set, and so on. Memory block 4 again maps to the first set in the cache.

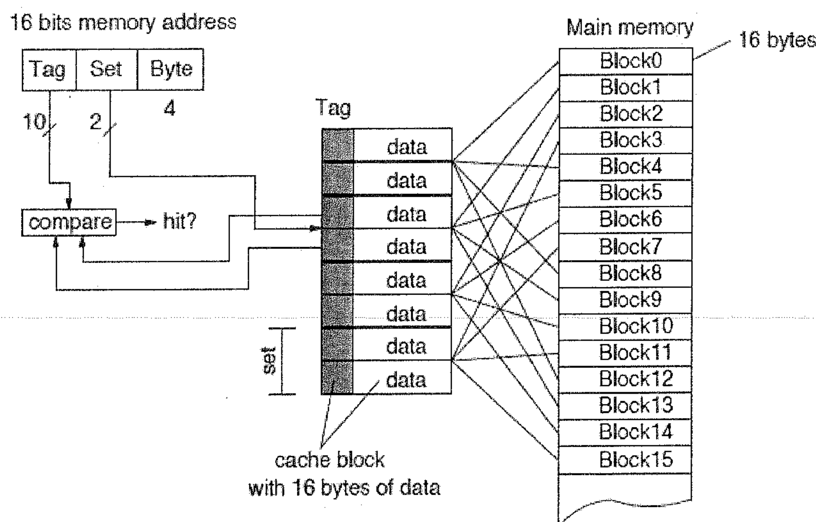


Figure 13.4: Set associative cache

In the example of figure 13.5, the least-significant bits are still used to address the byte in the cache block. The next 2 bits determine in which cache set the data should reside. To find out if there is a cache hit, the tags from all cache blocks in the set are compared in parallel with the 10 tag-bits provided by the processor. Compared to a fully associative solution, the set-associative approach reduces the number of comparisons that need to be performed in parallel as it now equals to the number of cache blocks inside a set. Studies have shown that 4-way set-associative caches already approach the performance of fully associative caches while being less expensive. Set-associative caches are therefore a popular choice for 1 -level cache implementations (these on-chip caches can only be of moderate size and must yield a good hit ratio).

13.2.4 Cache strategies

Besides a mapping mechanism, caches also need a range of strategies that specify what should happen in the case of certain events. In this section, we discuss several of these cache strategies.

In (set-)associative caches, the cache must determine which cache block is replaced by a new block entering the cache. There are several well-known replacement strategies, which are also applied in other fields such as in operating systems: Random, First-In First Out (FIFO) and Least Recently Used (LRU). Many caches apply some sort of LRU replacement (sometimes a type of pseudo-LRU instead of real LRU). Studies have shown that in most cases the LRU strategy performs best. Several other cache strategies relate to write actions. For example, when the processor performs a store instruction, the cache can follow one of two write policies. First, it can write the value of the store instruction into the cache *and* into the main memory. This is called a *write-through* cache since it writes the value through the cache to the memory. Alternatively, the cache can only update the cache at a write action. This is called a *write-back* cache.

The advantage of a write-through cache is that the main memory is always consistent with the cache (later on we will see why this may be useful). The disadvantage is, however, that it increases bus/memory traffic since every write action is propagated to the main memory. This is not done in a write-back cache, which is therefore more efficient in terms of memory traffic. However, such a cache needs an extra status bit per cache block specifying whether a cache block is dirty (written to) or not. In the event a dirty cache block is removed from the cache (e.g., it has been selected by the replacement strategy), it cannot be simply discarded but it needs to be written back to memory. Another potential drawback of write-back caches is that context switches of the OS may be slower than is the case with a write-through cache. Because switching to a new process often implies that a different range of addresses is used, a context switch may trigger a large number of replacements in the cache. With a write-back cache, this means that these replacements may lead to heavy memory traffic when many of the replaced blocks are dirty and need to be written back to memory. In a write-through cache, on the other hand, replacements are cheap: simply discard the cache block.

Another strategy related to write events is the 'write-miss strategy'. In other words, what should the cache do when a cache miss for a store instruction occurs? In the *allocate-on-write* strategy, the cache allocates a cache block in the cache after which it performs the write action on this block. When the cache implements the *fetch-on-write* strategy, the cache allocates a cache block *and* fetches the data that belongs to this cache block from main memory before performing the write action. Write-back caches often use one of these two write-miss strategies. Write-through caches may apply the *no-allocate-on-write* strategy, in which nothing is done on a write miss. The value is simply written to the memory and the cache contents remain unchanged. In this chapter, we have presented a brief overview of (issues relating to) high-performance processors applied in modern parallel systems. Again, if you are interested in a more in-depth discussion on microprocessor architecture, then you are referred to the Advanced Computer Architecture course. The next building block of parallel systems we discuss is the interconnection network which ties the processors together. The network determines the connectivity between processors and therefore has significant impact on the number of processors that can be accommodated by a parallel system.