# Beter, Sneller, Mooier

Processoren

27 maart 2012

# Beter! Sneller!

- Krachtigere CPU: microcode
- Snellere CPU: pipeline, out-of-order execution
- Sneller RAM: cache

# meer mogelijkheden...

- Welke extra's kan processor-bouwer bieden?
  - kleinere programma's
    → instructies die meer doen
  - snellere programma's
    → hogere klokfrequentie

- vandaag: een paar technieken daarvoor
  - niet in de practicum-processor gebruikt

# instructies die meer doen...

- bv. PUSH en POP bij de practicum-processor
  - schrijft naar geheugen/leest uit geheugen
  - verandert stack pointer

- microcode:
  - instructie uitvoeren als reeks van micro-stappen in een micro(machine)code
  - complexe instructies worden mogelijk zonder eenvoudige langzamer te maken

# Een voorbeeld-microarchitectuur

MBR = instructieregister

bepaalt MPC = microprogram counter

bepaalt MIR = microinstructieregister

Bits van het MIR
sturen onderdelen van processor aan.

B = welk register lezen?

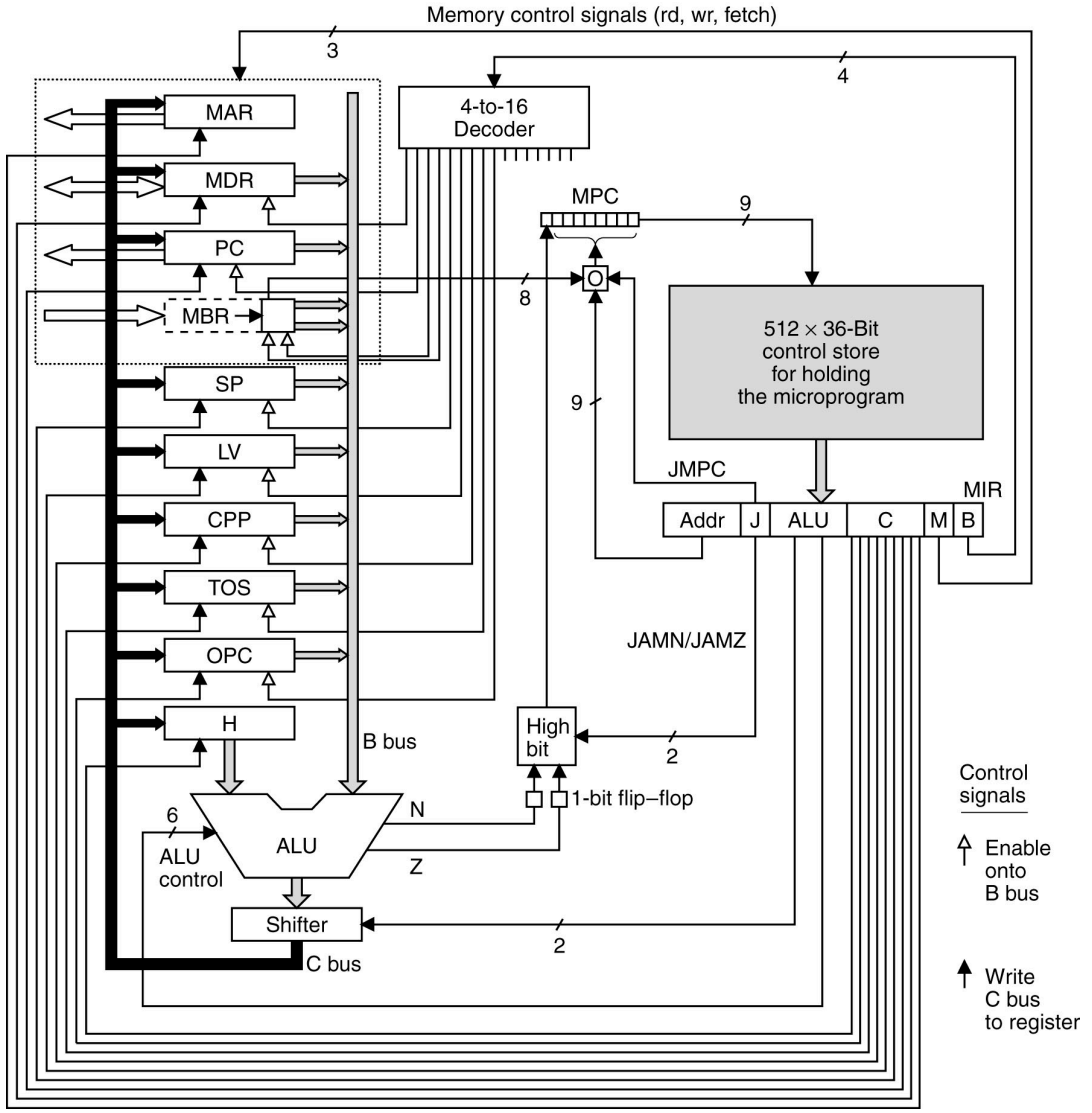
M = RAM-operatie kiezen

C = welk(e) register(s) schrijven?

ALU = welke ALU-operatie?

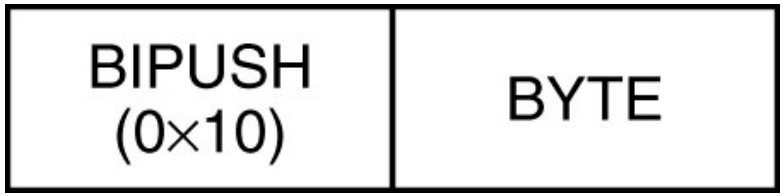
J = voorwaarden voor sprong

Addr = adres van volgende microinstructie



# voorbeeldinstructie: BIPUSH = constante op stack zetten

- instructieformaat: 2 bytes




- microprogramma:
  hiervoor werd MBR al op BYTE gezet
  stackgeheugen: lage adressen zijn in gebruik

```
SP = MAR = SP + 1          MBR = the byte to push onto stack
PC = PC + 1; fetch         Increment PC, fetch next opcode
MDR = TOS = MBR; wr; goto Main1   Sign-extend constant and push on stack
```

# Krachtigere CPU

- microcode

# Pipeline

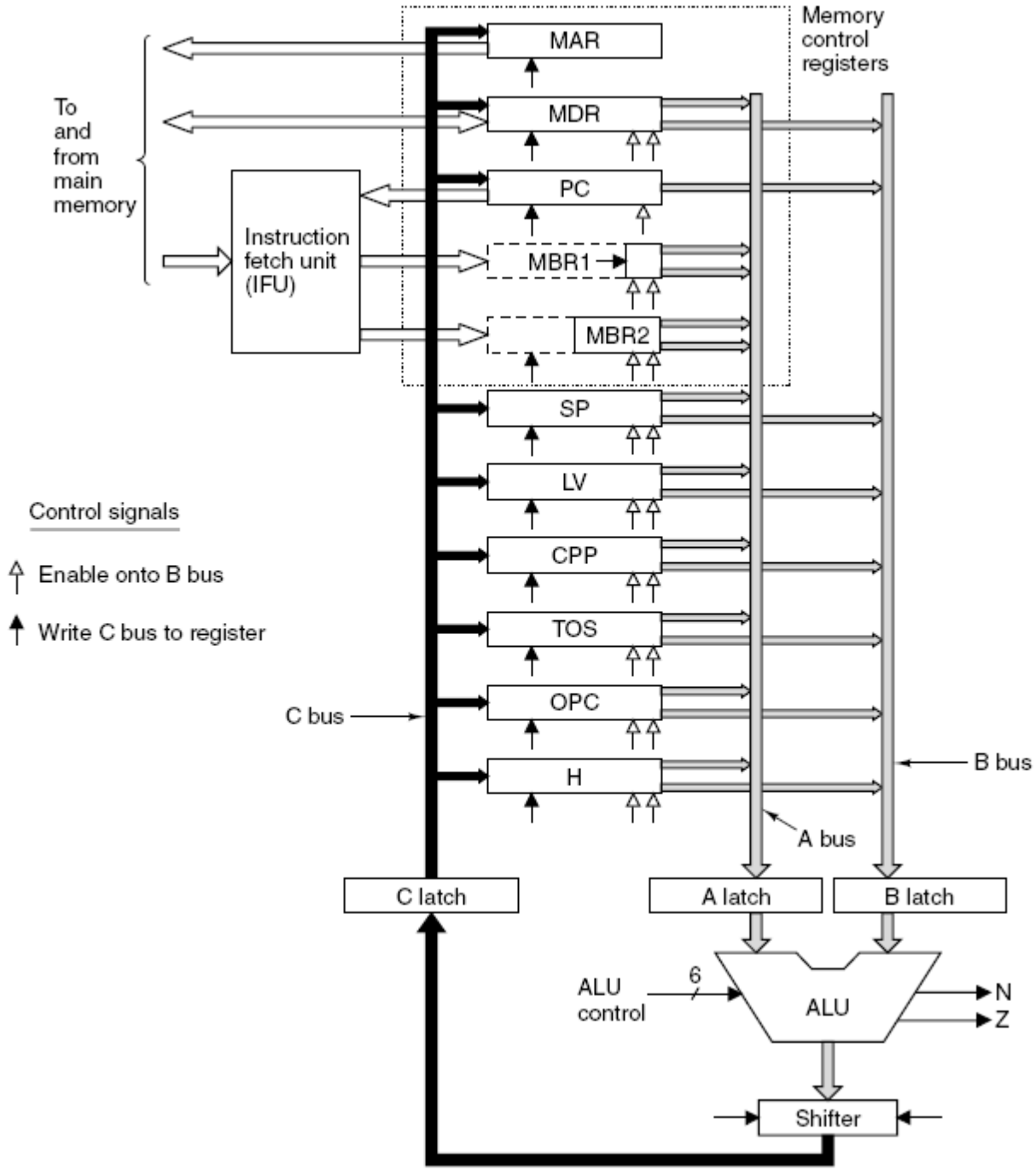
- al genoemd: 3 stappen
  fetch → decode → execute
- practicum-processor: 2 stappen
  fetch en PC verhogen → decode en execute
- moderne processoren: tot ±20 stappen

- voordeel:
  eenvoudige stappen → hoge kloksnelheid

# Three Step Architecture

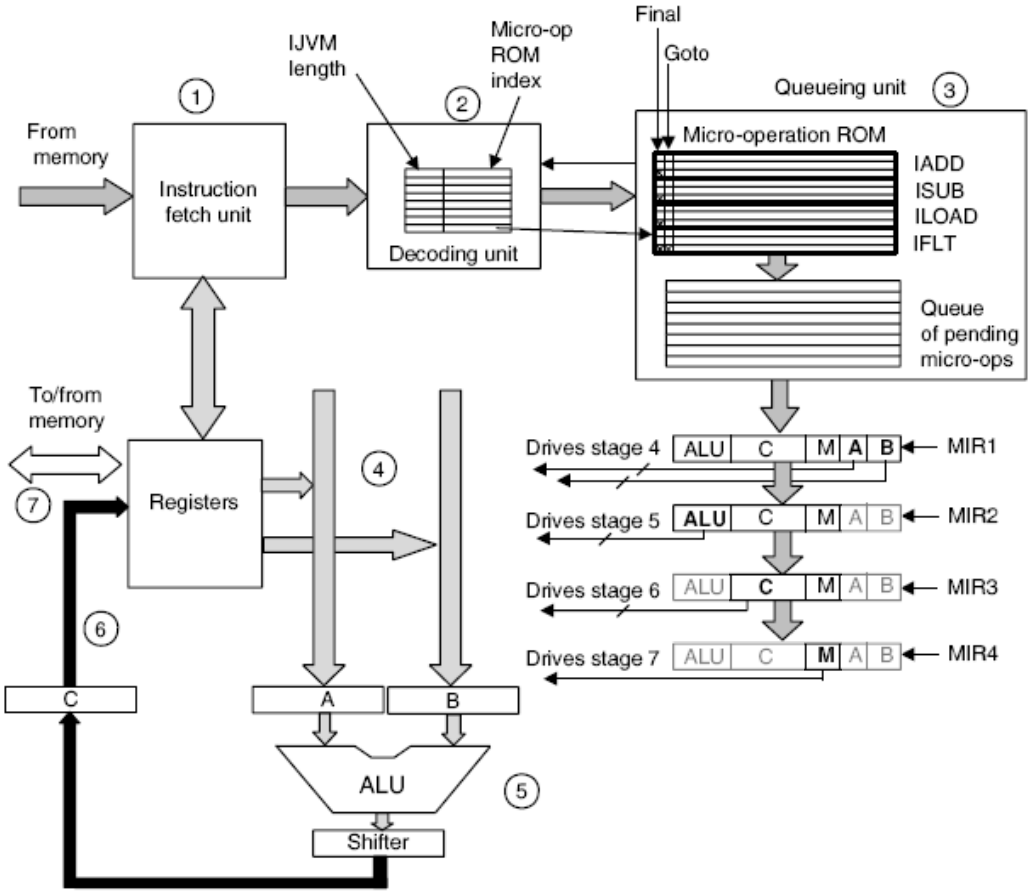
The data path
used in the Mic-3:

Every calculation takes
three cycles.



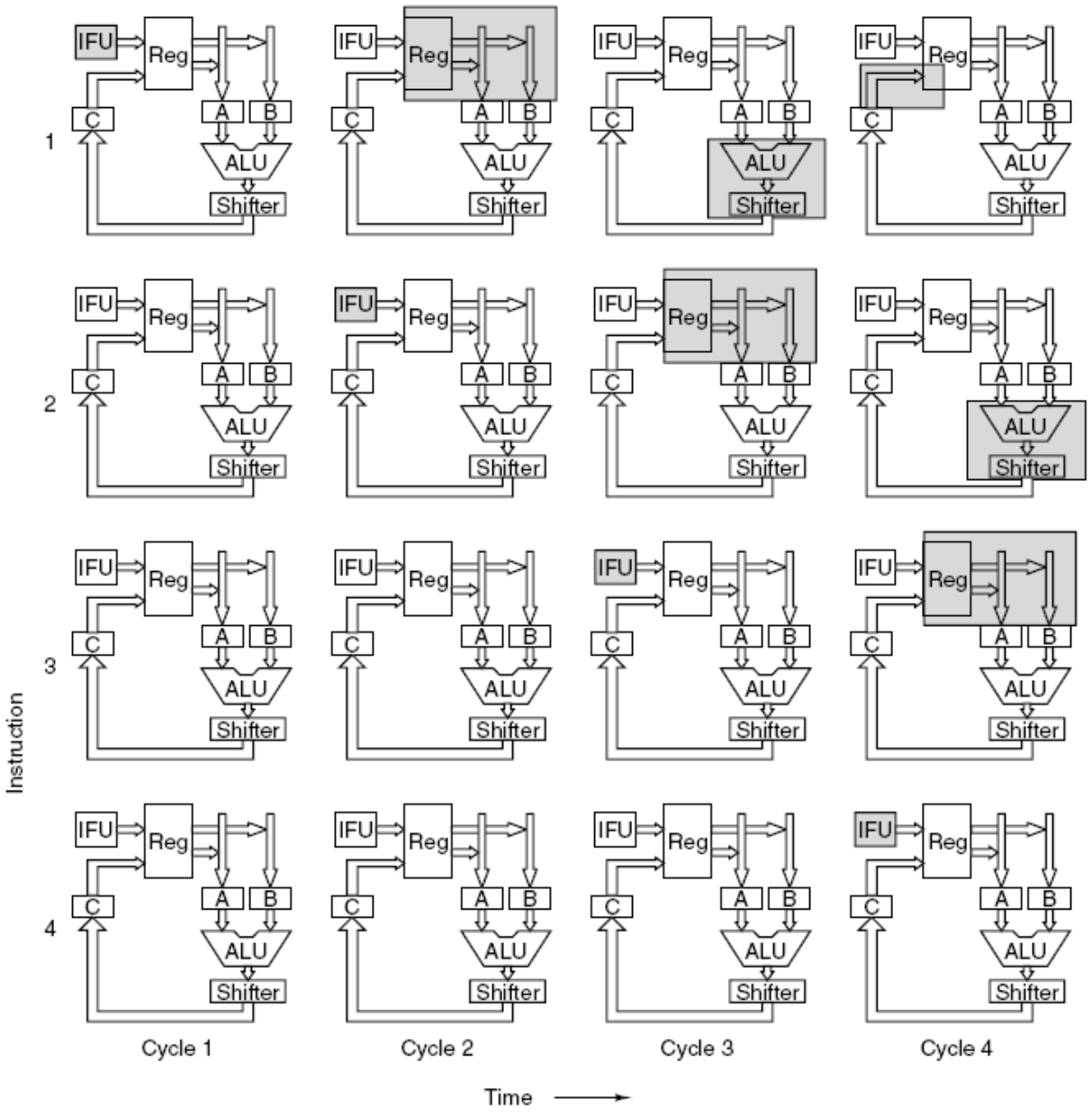
# De voorbeeldarchitectuur met (lange) pipeline

1. fetch: instructie lezen
2. decode: bepaal lengte + kies microcode
3. queue: plaats microcode-stappen in wachtrij
4. lees registers
5. voer berekening uit
6. schrijf registers
7. lees/schrijf RAM



# Pipeline

Graphical illustration
of how a pipeline works.



# voorbeeldinstructie: SWAP = twee elementen op stack verwisselen

- microcode in 1e versie:

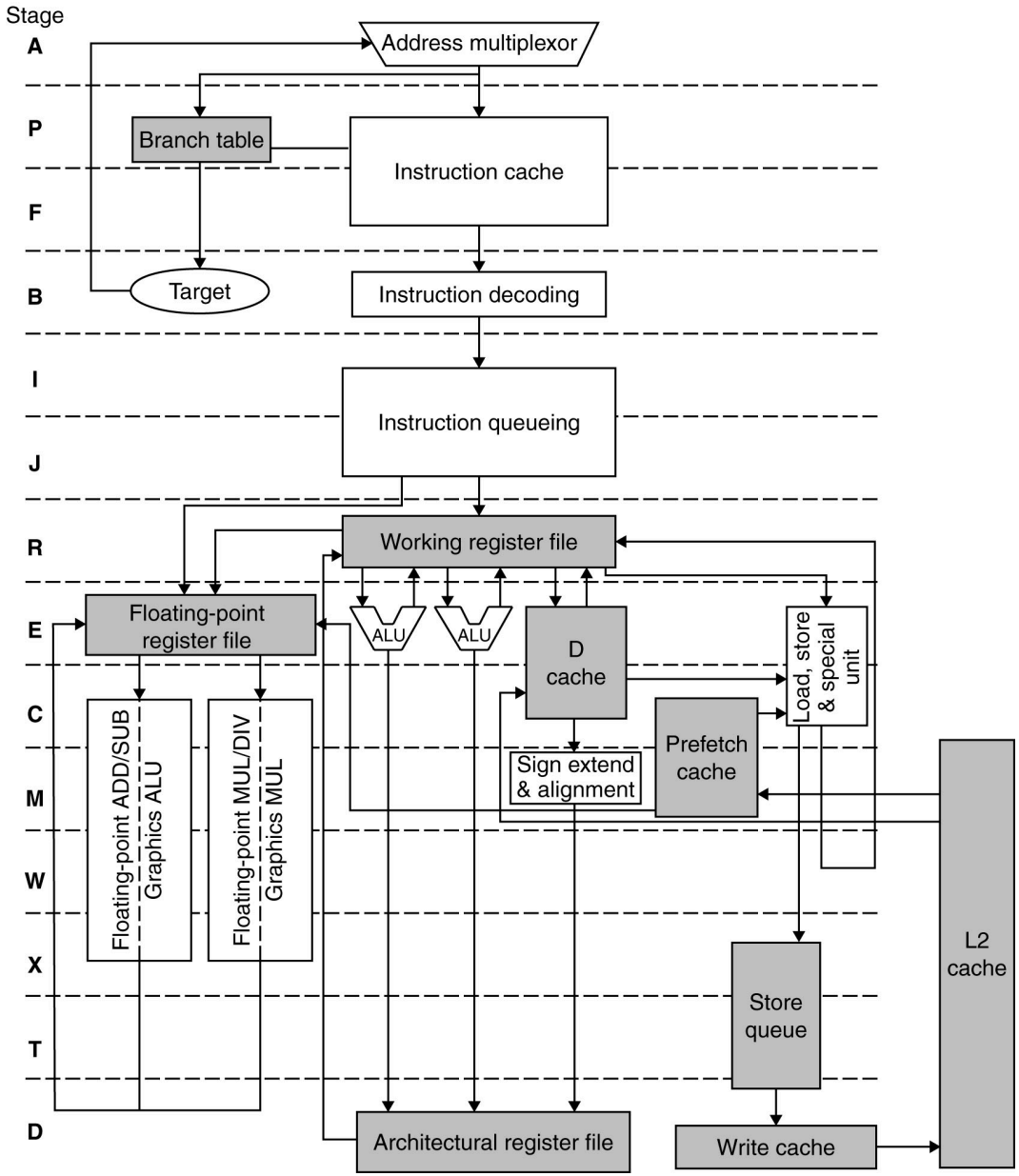
| Label | Operations | Comments |
|---|---|---|
| swap1 | MAR = SP – 1; rd | Read 2nd word from stack; set MAR to SP |
| swap2 | MAR = SP | Prepare to write new 2nd word |
| swap3 | H = MDR; wr | Save new TOS; write 2nd word to stack |
| swap4 | MDR = TOS | Copy old TOS to MDR |
| swap5 | MAR = SP – 1; wr | Write old TOS to 2nd place on stack |
| swap6 | TOS = H; goto Main1 | Update TOS |

- nieuwe microcode:
  elke stap in 3 kleine stapjes opdelen

# SWAP met pipeline

| | Swap1 | Swap2 | Swap3 | Swap4 | Swap5 | Swap6 |
|---|---|---|---|---|---|---|
| Cy | MAR=SP–1;rd | MAR=SP | H=MDR;wr | MDR=TOS | MAR=SP–1;wr | TOS=H;goto (MBR1) |
| 1 | B=SP | | | | | |
| 2 | C=B–1 | B=SP | | | | |
| 3 | MAR=C; rd | C=B | | | | |
| 4 | MDR=Mem | MAR=C | | | | |
| 5 | | | B=MDR | | | |
| 6 | | | C=B | B=TOS | | |
| 7 | | | H=C; wr | C=B | B=SP | |
| 8 | | | Mem=MDR | MDR=C | C=B–1 | B=H |
| 9 | | | | | MAR=C; wr | C=B |
| 10 | | | | | Mem=MDR | TOS=C |
| 11 | | | | | | goto (MBR1) |

# Pipeline voorbeeld: SPARC



# GOTO statement considered harmful

- Bij sprong-instructies zijn er twee mogelijke voortzettingen van het programma
- welke van de twee moet de processor nemen – pas bekend als de vorige instructie klaar is
- pipeline raakt leeg → inefficiënt!

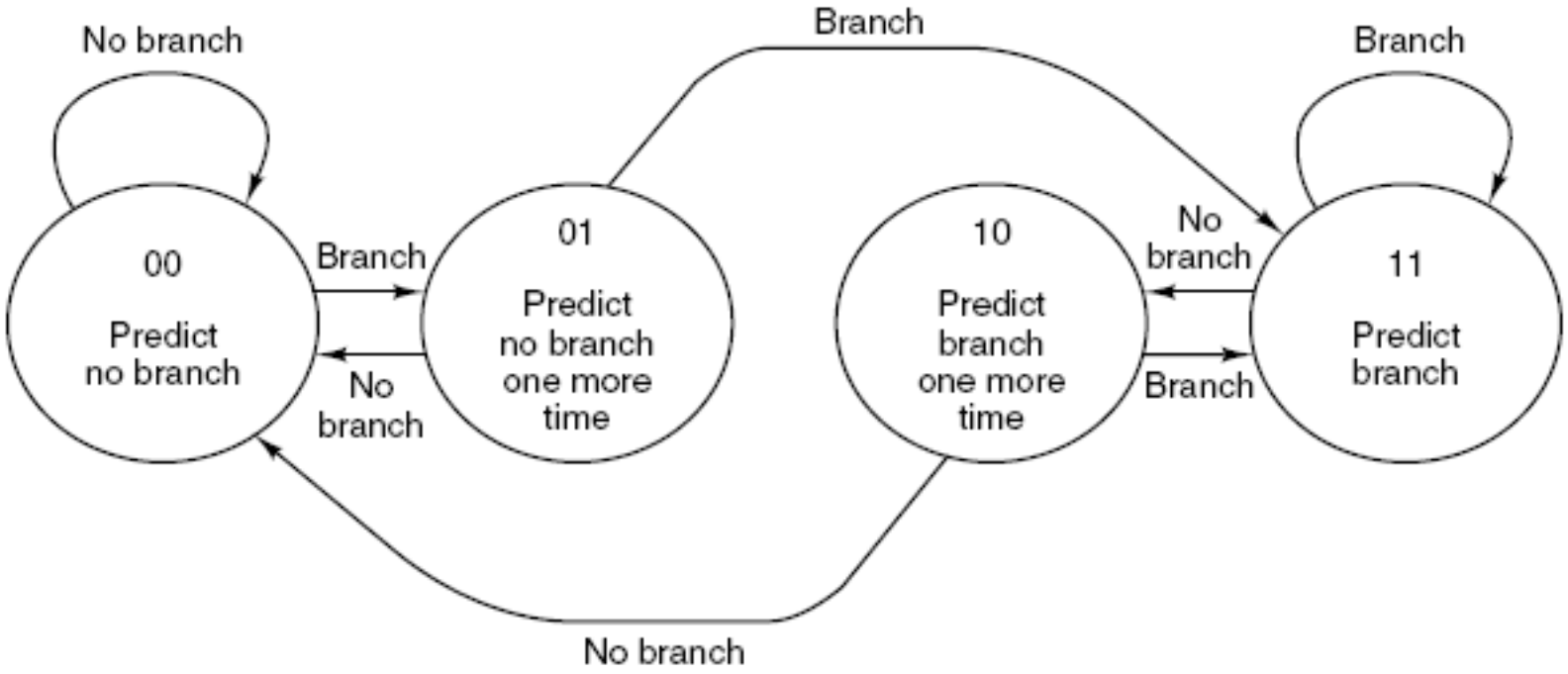
- trucje bij SPARC: sprongen werken met één instructie vertraging

# Oplossing: branch prediction

- branch prediction spreekt een **vermoeden** uit over de waarschijnlijke voortzetting

- statisch: b.v. „spring achteruit (loop), maar niet vooruit (if)”
- dynamisch: „spring zoals de vorige twee keer”
  - (gebruikt branch prediction cache)

# dynamische branch prediction




voorbeeld van een toestandsmachine

# out-of-order execution

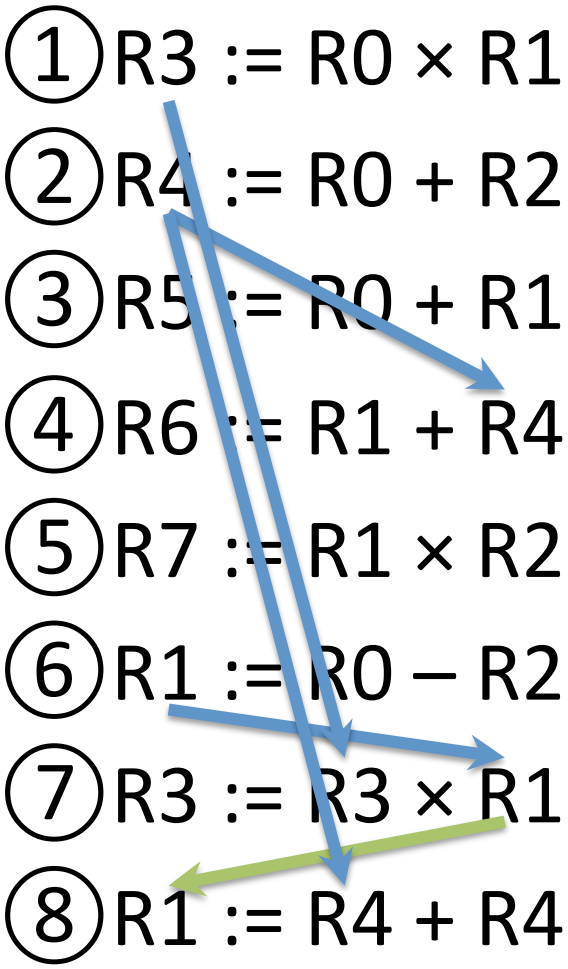
- sommige programma's kiezen onhandige volgorde van instructies
  - instructies met data-afhankelijkheden direct achter elkaar
- processor „verbetert" de volgorde waar mogelijk
  - instructies zonder data-afhankelijkheden alvast starten

# register renaming

- sommige data-afhankelijkheden zijn niet echt
  - WAR: write after read
  - WAW: write after write
- gebruiken hetzelfde register,
  maar tweede instructie gebruikt
  geen resultaat van eerste

- oplossing: schaduwregisters

# out-of-order execution: voorbeeld

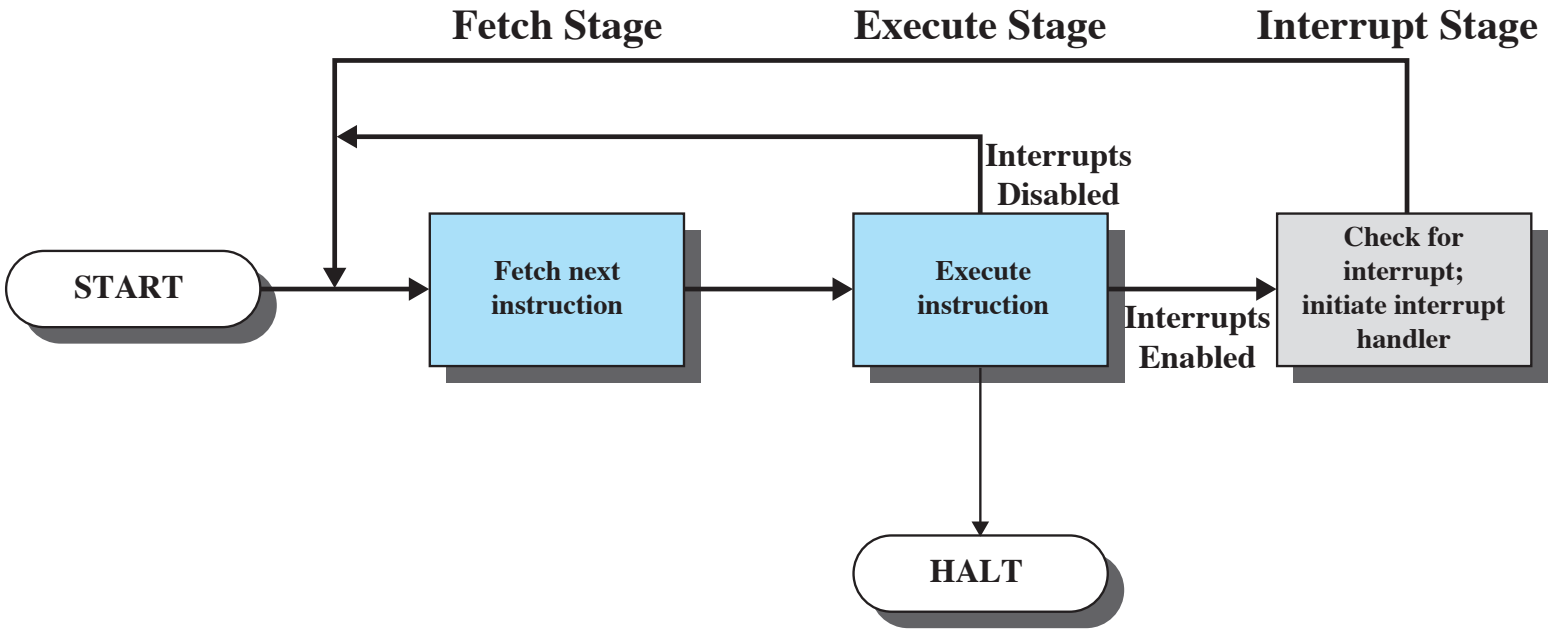


# out-of-order execution: voorbeeld

| | | | | | Registers being read | | | | | | | | Registers being written | | | | | | | | Registers being stalled | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cy | # | Decoded | Iss | Ret | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | R3=R0*R1 | 1 | | 1 | 1 | | | | | | | | | | 1 | | | | | | | | | | | | |
| | 2 | R4=R0+R2 | 2 | | 2 | 1 | 1 | | | | | | | | | 1 | 1 | | | | | | | | | | | |
| 2 | 3 | R5=R0+R1 | 3 | | 3 | 2 | 1 | | | | | | | | | 1 | 1 | 1 | | | | | | | | | | |
| | 4 | R6=R1+R4 | – | | 3 | 2 | 1 | | | | | | | | | 1 | 1 | 1 | | | | | | | | | 1 | |
| 3 | 5 | R7=R1*R2 | 5 | | 3 | 3 | 2 | | | | | | | | | 1 | 1 | 1 | | 1 | | | | | | | 1 | |
| | 6 | S1=R0–R2 | 6 | | 4 | 3 | 3 | | | | | | | | | 1 | 1 | 1 | | 1 | | | | | | | 1 | |
| | | | | 2 | 3 | 3 | 2 | | | | | | | | | 1 | | 1 | | 1 | | | | | | | 1 | |
| 4 | | | 4 | | 3 | 4 | 2 | | 1 | | | | | | | 1 | | 1 | 1 | 1 | | | | | | | | |
| | 7 | R3=R3*S1 | – | | 3 | 4 | 2 | | 1 | | | | | | | 1 | | 1 | 1 | 1 | | | | 1 | | | | |
| | 8 | S2=R4+R4 | 8 | | 3 | 4 | 2 | | 3 | | | | | | | 1 | | 1 | 1 | 1 | | | | 1 | | | | |
| | | | | 1 | 2 | 3 | 2 | | 3 | | | | | | | | | 1 | 1 | 1 | | | | 1 | | | | |
| | | | | 3 | 1 | 2 | 2 | | 3 | | | | | | | | | | 1 | 1 | | | | 1 | | | | |
| 5 | | | | 6 | | 2 | 1 | | 3 | | | | | 1 | | | | | 1 | 1 | | | | 1 | | | | |
| 6 | | | 7 | | | 2 | 1 | 1 | 3 | | | | | 1 | | 1 | | | 1 | 1 | | | | | | | | |
| | | | | 4 | | 1 | 1 | 1 | 2 | | | | | 1 | | 1 | | | | 1 | | | | | | | | |
| | | | | 5 | | | | 1 | 2 | | | | | 1 | | 1 | | | | | | | | | | | | |
| | | | | 8 | | | | 1 | | | | | | | | 1 | | | | | | | | | | | | |
| 7 | | | | | | | | 1 | | | | | | | | 1 | | | | | | | | | | | | |
| 8 | | | | | | | | 1 | | | | | | | | 1 | | | | | | | | | | | | |
| 9 | | | | 7 | | | | | | | | | | | | | | | | | | | | | | | | |

# precise interrupt

- traditionele interrupt:
  precies tussen twee instructies



# precise interrupt

- traditionele interrupt:
  precies tussen twee instructies
- pipeline: instructie-executies overlappen
- out-of-order execution: instructies te laat klaar
- oplossing: extra control logic
  om precise interrupt toch te realiseren
- algemeen: gedrag van CPU met pipeline
  mag niet afwijken van CPU zonder pipeline

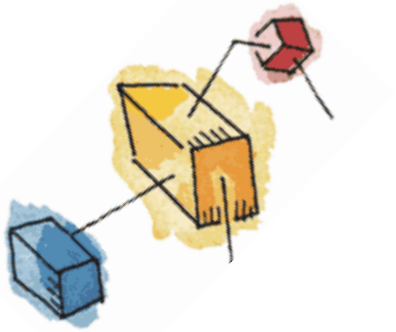
# Snellere CPU

- pipeline
- branch prediction
- out-of-order execution
- register renaming

- probleem: precise interrupt

# Cache Memory

- Invisible to the OS
  - Interacts with other memory management hardware
- Processor must access memory at least once per instruction cycle
  - Processor speed faster than memory access speed
- Exploit the principle of locality with a small fast memory



# Principle of Locality

- More details later but in short …
- Data which is required soon is often close to the current data
  - If data is referenced, then it's neighbour might be needed soon.

# Cache and Main Memory

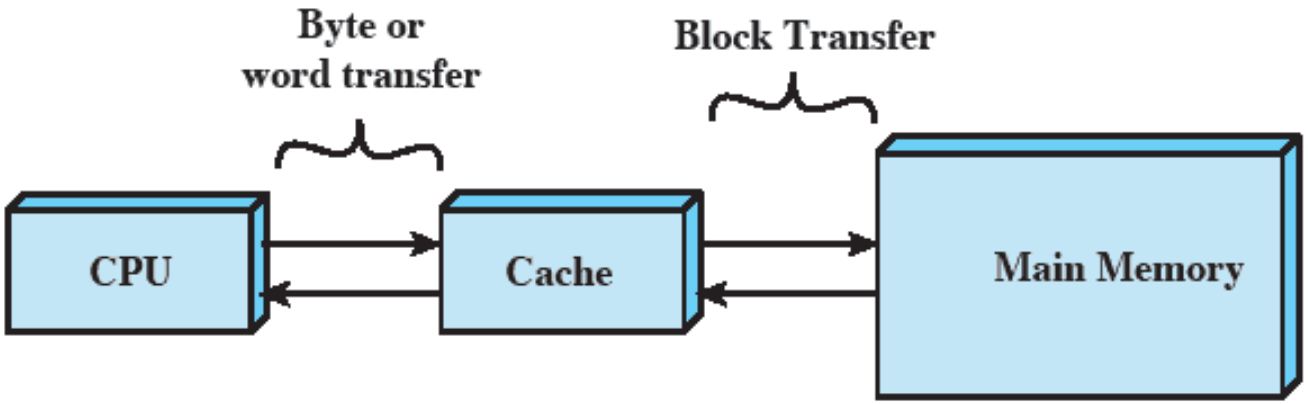



Figure 1.16 Cache and Main Memory

# Cache Principles

- Contains copy of a portion of main memory
- Processor first checks cache
  - If not found, block of memory read into cache
- Because of locality of reference, likely future memory references are in that block

# Cache/Main-Memory Structure

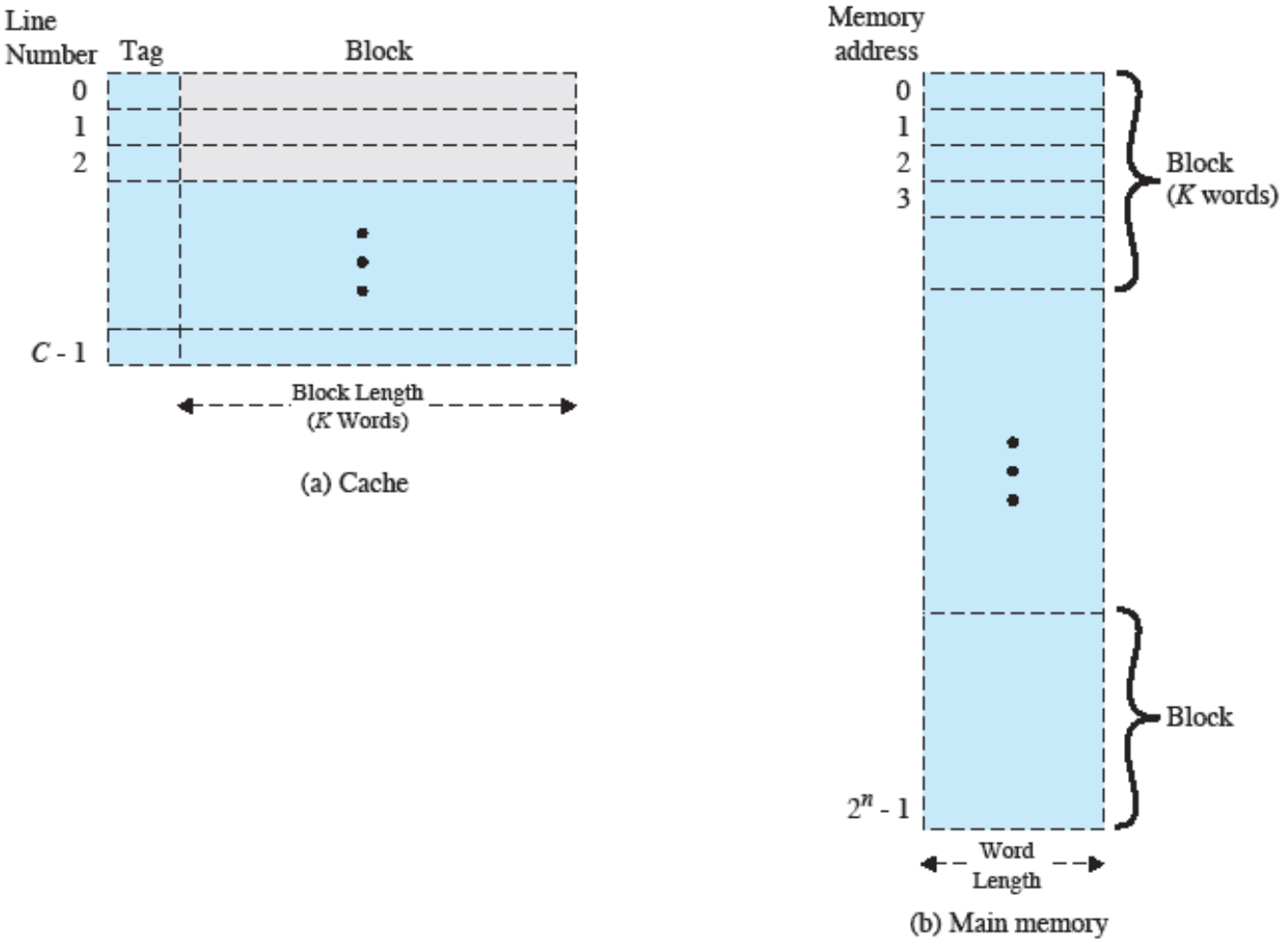


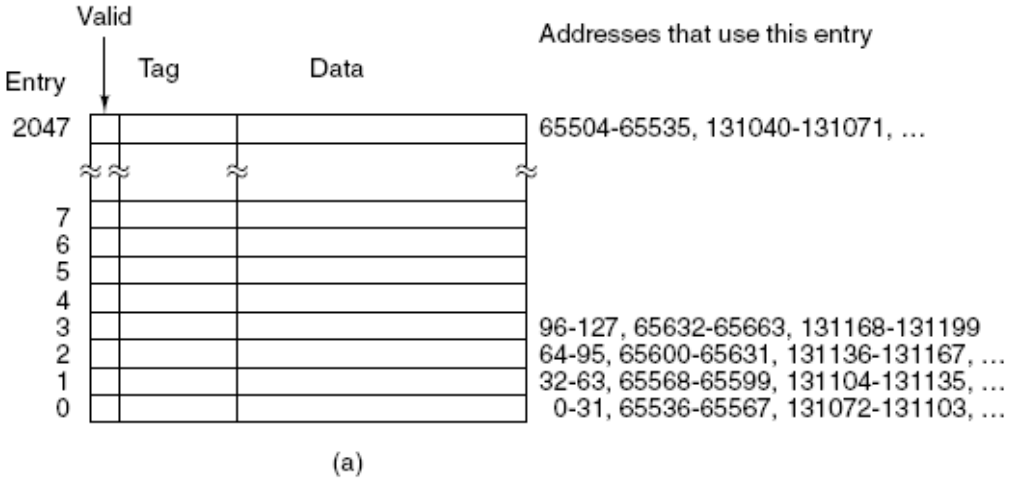

**Figure 1.17 Cache/Main-Memory Structure**

# direct-mapped cache

(a) cache-structuur

Elk data-veld bevat 8 words
van 4 bytes, dus 32 bytes.

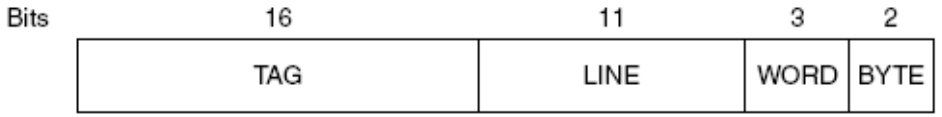
In elke cache-regel kan één blok
van 32 bytes opgeslagen worden.
De tag geeft aan welke.




(a)

(b) bitpatroon van een adres (32 bits)

In een adres geeft “line” aan
in welke regel van de cache
de CPU moet zoeken.




(b)

# Sneller RAM

- cache