Lektion 6

Optimering af Mikroarkitektur

Emil Lykke Diget

Computerarkitektur og Operativsystemer
Syddansk Universitet

Introduction

Computerarkitektur og Operativsystemer

Viden

- Kombinatoriske logiske kredse
- Memorytyper
- Memoryinterface incl. timing
- Adressedekodning
- Interrupt og exceptions
- **Computerdesign**
- Register-transfer level
- **Datapath**
- Control unit
- **Instruktionssæt**
- **Pipeline**
- **Cache**
- Processer og tråde
- Context switch
- Inter-process synkronisering og kommunikation, kritiske sektorer og semaphores

Færdigheder

- Redegøre for principperne og algoritmerne bag operativsystemets centrale funktioner
- **Forstå opbygningen af en moderne CPU**
- Kende de almindeligt forekomne memorytyper
- Forstå centrale begreber omkring et operativsystems afvikling af et program

Kompetencer

- Implementere operativsystemsfunktioner i et RTOS (Real Time Operating System)

- Lektion 1: Kombinatoriske Logiske Kredse
- Lektion 2: En Simpel Computer
- Lektion 3: Hukommelse
- Lektion 4: Mikroarkitektur
- Lektion 5: Micro-assembly og IJVM
- Lektion 6: Optimering af Mikroarkitektur

Indhold

Indhold

Optimering af Mikroarkitekturdesignet

Der er mange kriterier at overveje, når man designer en mikroarkitektur, f.eks.,

- Hastighed
- Pris
- Brugervenlighed
- Energiforbrug
- Fysisk størrelse

Optimering af Mikroarkitekturdesignet

Der er mange kriterier at overveje, når man designer en mikroarkitektur, f.eks.,

- Hastighed
- Pris
- Brugervenlighed
- Energiforbrug
- Fysisk størrelse

De fleste vigtige beslutninger er på baggrund af

Hastighed vs. Pris

Hastighed vs. Pris

Prisen afhænger primært af chip-størrelsen

- Større kompleksitet $\rightarrow$ større chip $\rightarrow$ højere pris
- Mindre kompleksitet $\rightarrow$ mindre chip $\rightarrow$ lavere pris

Hastighed vs. Pris

Prisen afhænger primært af chip-størrelsen

- Større kompleksitet $\rightarrow$ større chip $\rightarrow$ højere pris
- Mindre kompleksitet $\rightarrow$ mindre chip $\rightarrow$ lavere pris

Optimering af mikrokode

- Optimal mikrode $\rightarrow$ reduktion i antal cykluser pr. IJVM-instruktion $\rightarrow$ større hastighed

Hastighed vs. Pris

Prisen afhænger primært af chip-størrelsen

- Større kompleksitet $\rightarrow$ større chip $\rightarrow$ højere pris
- Mindre kompleksitet $\rightarrow$ mindre chip $\rightarrow$ lavere pris

Optimering af mikrokode

- Optimal mikrode $\rightarrow$ reduktion i antal cykluser pr. IJVM-instruktion $\rightarrow$ større hastighed

Modifikation af data-path

- Mere fleksibel bus-design $\rightarrow$ flere mikroinstruktionsmuligheder $\rightarrow$ reduktion i antal cykluser pr. IJVM-instruktion

Hastighed vs. Pris

Prisen afhænger primært af chip-størrelsen

- Større kompleksitet → større chip → højere pris
- Mindre kompleksitet → mindre chip → lavere pris

Optimering af mikrokode

- Optimal mikrode → reduktion i antal cykluser pr. IJVM-instruktion → større hastighed

Modifikation af data-path

- Mere fleksibel bus-design → flere mikroinstruktionsmuligheder → reduktion i antal cykluser pr. IJVM-instruktion

Andre hardware-ændringer

- Prefetching → reduktion i antal cykluser pr. IJVM-instruktion → større hastighed
- Pipelining

Merge af main-løkken og mikrokoden.
Før optimering:

Label	Operationer	Kommentar
pop1	MAR = SP = SP - 1; rd	Dekrementer SP og læs
pop2		Vent til ny TOS er læst fra hukommelsen
pop3	TOS = MDR; goto Main1	Kopier nyt word til TOS
Main1	PC = PC + 1; fetch; goto (MBR)	

- pop2 bruges kun til at vente på, at MDR opdateres.
- POP-instruktionen tager fire clock-cykler fra den startes, til den næste IJVM-instruktion køres.

Merge af main-løkken og mikrokoden.

Label	Operationer	Kommentar
pop1	MAR = SP = SP - 1; rd	Dekrementer SP og læs
Main1.pop	PC = PC + 1; fetch	MBR holder opcode, skaf næste byte
pop3	TOS = MDR; goto (MBR)	Kopier nyt word til TOS, gå til næste instruktion

- Main flyttes ind i POP, så den kun tager tre clock-cykler i stedet for fire.

En besparelse på 25% hver gang, POP-instruktionen køres.

Modifikation af Data-Path

Tre-bus-arkitektur. Eksempel: ILOAD

Gemmer lokal variabel nr. *index* (unsiged) på stakken.

ILOAD	INDEX
0x15	

Original mikrokdoe:

Label	Operationer	Kommentar
iload1	H = LV	MBR indeholder index. Kopier LV til H
iload2	MAR = MBRU + H; rd	Få global adresse på den lokale variable
iload3	MAR = SP = SP + 1	SP peger på den nye stak-top
iload4	PC = PC + 1; fetch; wr	Inkrementer PC og skaf næste opcode. Skriv til stak-top
iload5	TOS = MDR; goto Main1	Opdater TOS
Main1	PC = PC + 1; fetch; goto (MBR)	

Modifikation af Data-Path

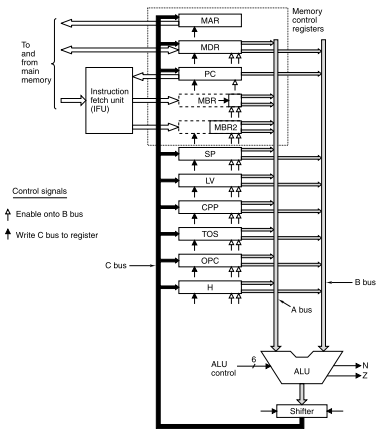
Tre-bus-arkitektur. Eksempel: ILOAD

Gemmer lokal variabel nr. *index* (unsiged) på stakken.

Mikrokode med 3-bus-arkitektur.

Label	Operationer
iload1	MAR = MBRU + LV; rd
iload2	MAR = SP = SP + 1
iload3	PC = PC + 1; fetch; wr
iload4	TOS = MDR
iload5	PC = PC + 1; fetch; goto (MBR)

Med en ekstra bus (A-bussen), er flere operationer lovlige.



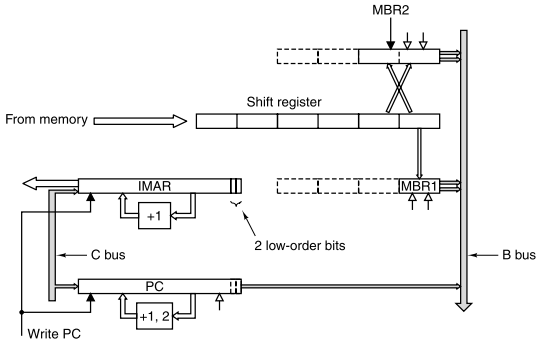
Andre Hardware-ændringer

Instruction Fetch Unit (IFU)

I Mic-1 bruges ALUen ofte til at forøge PC, og andet relateret til instruktioner og operander.

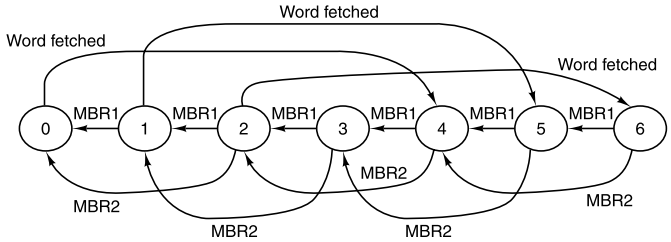
Instruction fetch unit (IFU) skaffer automatisk opkoder og operander.

- Opkoder i MBR1 (8-bit).
- Operander i MBR2 (16-bit).
- Instruction Memory Adress Register (IMAR) skaffer instruktioner fra metode-området 4 byte af gangen.
- PC peger altid til den første byte, der endnu ikke er brugt.



Andre Hardware-ændringer

Instruction Fetch Unit (IFU) – Finite State Machine



Transitions

MBR1: Occurs when MBR1 is read
MBR2: Occurs when MBR2 is read
Word fetched: Occurs when a memory word is read and 4 bytes are put into the shift register

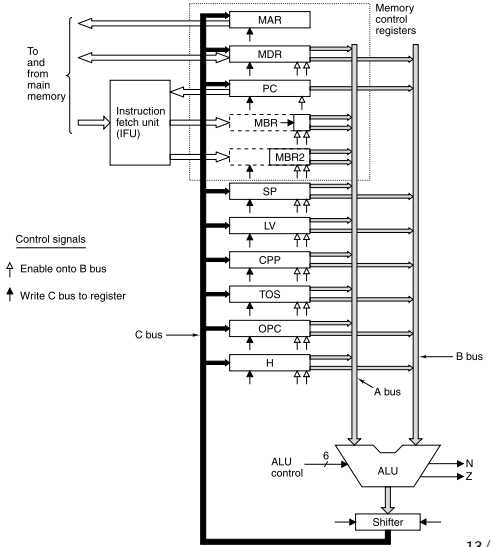
- ($\leftarrow$) indikerer læse fra MBR1 (8-bit) eller MBR2 (16-bit).
- ($\rightarrow$) indikerer skrive til skifteregisteret (32-bit).

Indhold

Mic-2

Hurtigere end Mic-1, men også mere kompliceret.

- Ingen main-løkke.
- Inkrementering af PC er ikke længere ALUens opgave.
- Næste instruktion (i MBR1) er altid klar.
- 16-bit index/offset genereres direkte (MBR2).



Mic-2-mikrokode

Label	Operations	Comments
nop1	goto (MBR)	Branch to next instruction
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR+H; wr; goto (MBR1)	Add top two words; write to new top of stack
isub1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR–H; wr; goto (MBR1)	Subtract TOS from Fetched TOS-1
iand1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto (MBR1)	AND Fetched TOS-1 with TOS
ior1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto (MBR1)	OR Fetched TOS-1 with TOS
dup1	MAR = SP = SP + 1	Increment SP; copy to MAR
dup2	MDR = TOS; wr; goto (MBR1)	Write new stack word
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for read
pop3	TOS = MDR; goto (MBR1)	Copy new word to TOS

Mic-2-mikrokode

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 (MBR1)	Update TOS
bipush1	SP = MAR = SP + 1	Set up MAR for writing to new top of stack
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Update stack in TOS and memory
iload1	MAR = LV + MBR1U; rd	Move LV + index to MAR; read operand
iload2	MAR = SP = SP + 1	Increment SP; Move new SP to MAR
iload3	TOS = MDR; wr; goto (MBR1)	Update stack in TOS and memory
istore1	MAR = LV + MBR1U	Set MAR to LV + index
istore2	MDR = TOS; wr	Copy TOS for storing
istore3	MAR = SP = SP – 1; rd	Decrement SP; read new TOS
istore4		Wait for read
istore5	TOS = MDR; goto (MBR1)	Update TOS
wide1	goto (MBR1 OR 0x100)	Next address is 0x100 ored with opcode
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Identical to iload1 but using 2-byte index
wide_istore1	MAR = LV + MBR2U; goto istore2	Identical to istore1 but using 2-byte index
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	Same as wide_iload1 but indexing off CPP

Mic-2-mikrokode

iinc1	MAR = LV + MBR1U; rd	Set MAR to LV + index for read
iinc2	H = MBR1	Set H to constant
iinc3	MDR = MDR + H; wr; goto (MBR1)	Increment by constant and update
goto1	H = PC – 1	Copy PC to H
goto2	PC = H + MBR2	Add offset and update PC
goto3		Have to wait for IFU to fetch new opcode
goto4	goto (MBR1)	Dispatch to next instruction
iflt1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit
ifeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit
if_icmpeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP – 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = H – OPC; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F

Mic-2-mikrokode

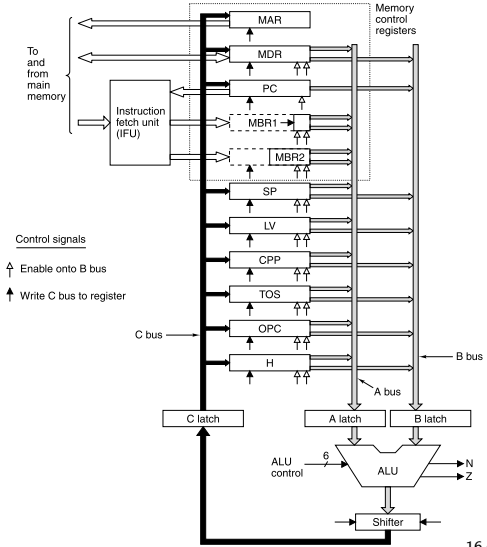
T	H = PC – 1; goto goto2	Same as goto1
F	H = MBR2	Touch bytes in MBR2 to discard
F2	goto (MBR1)	
invokevirtual1	MAR = CPP + MBR2U; rd	Put address of method pointer in MAR
invokevirtual2	OPC = PC	Save Return PC in OPC
invokevirtual3	PC = MDR	Set PC to 1st byte of method code.
invokevirtual4	TOS = SP – MBR2U	TOS = address of OBJREF – 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = address of OBJREF
invokevirtual6	MDR = SP + MBR2U + 1; wr	Overwrite OBJREF with link pointer
invokevirtual7	MAR = SP = MDR	Set SP, MAR to location to hold old PC
invokevirtual8	MDR = OPC; wr	Prepare to save old PC
invokevirtual9	MAR = SP = SP + 1	Inc. SP to point to location to hold old LV
invokevirtual10	MDR = LV; wr	Save old LV
invokevirtual11	LV = TOS; goto (MBR1)	Set LV to point to zeroth parameter.
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to read Link ptr
ireturn2		Wait for link ptr
ireturn3	LV = MAR = MDR; rd	Set LV, MAR to link ptr; read old PC
ireturn4	MAR = LV + 1	Set MAR to point to old LV; read old LV
ireturn5	PC = MDR; rd	Restore PC
ireturn6	MAR = SP	
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Save return value on original top of stack

Indhold

Data-pathen i Mic-3

Mic-3s data-path er delt op i tre:

- Tre latches er placeret i midten af hver bus.
- Latches opdateres i hver cyklus.
- Clock-hastigheden er nu tre gange hurtigere, og der er mindre forsinkelse per del.
- Hver del kan arbejde på forskellige dele af IJVM-instruktionen samtidigt.



SWAP-instruktionen for Mic-2 tager seks mikroinstruktioner.

Label	Operationer	Kommentar
swap1	MAR = SP - 1; rd	Læs 2. word fra toppen af stakken
swap2	MAR = SP	Sæt MAR til toppen af stakken
swap3	H = MDR; wr	Gem ny TOS i H og skriv 2. word fra toppen af stakken
swap4	MDR = TOS	Kopier gammel TOS til MDR
swap5	MAR = SP - 1; wr	Skriv gammel TOS til 2. word fra toppen af stakken
swap6	TOS = H; goto (MBR1)	Opdater TOS og gå til næste opcode

Pga. latches i Mic-3, kan hver mikroinstruktion deles op i tre mikroskridt.

$$1\,\text{mikroinstruktion} = 3\,\Delta T$$

Pga. latches i Mic-3, kan hver mikroinstruktion deles op i tre mikroskridt.

$$1\,\text{mikroinstruktion} = 3\,\Delta\text{T}$$

Sagt på en anden måde:

> *Én mikroinstruktioncyklus i Mic-2 er tre cyklusser i Mic-3.*

For Mic-2 tager SWAP $18\,\Delta\text{T}$.

Mic-3-eksempel

SWAP

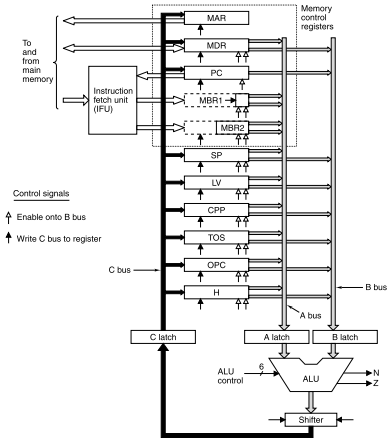
- Vi kan starte en mikroinstruktion efter hver cyklus. Næsten!
- Read After Write-afhængighed (RAW).
  - Et mikrostep forsøger at læse et register, der endnu ikke er skrevet til.
  - rd starter i cyklus 3, så vi kan læse den i cyklus 5.
  - Pipeline stall på 2 cykluser.

Swap tager 11 ΔT modsat 18 ΔT for Mic-2.

	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)

Mic-3-eksempel

SWAP

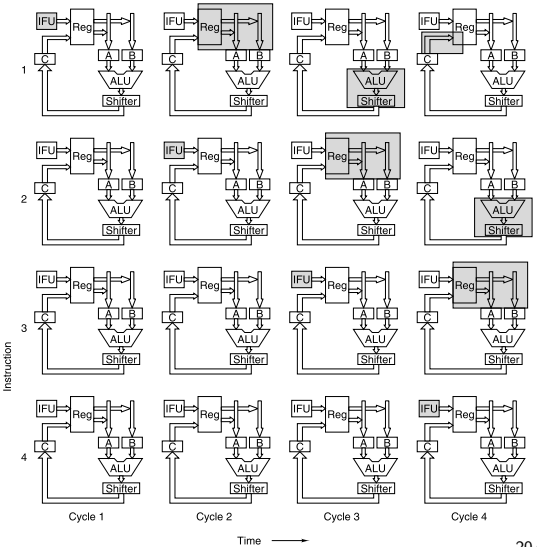


	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
Cy	**MAR=SP–1;rd**	**MAR=SP**	**H=MDR;wr**	**MDR=TOS**	**MAR=SP–1;wr**	**TOS=H;goto (MBR1)**
1	B=SP					
2	C=B–1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=Mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B–1	B=H
9					MAR=C; wr	C=B
10	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

Indhold

Pipeline – Mic-3

- Pipeline for Mic-3 uden stalls.
- Fire-stadie pipeline.
- Hver instruktion tager fire clock-cykler at udføre.
- I hver cyklus startes en ny instruktion.
- I hver cyklus er en instruktion færdig.



En instruktion kan deles op i fem faser

S1: Fetch af instruktion.
Instruktionen loades fra hukommelsen ind i CPUen.

S2: Dekodning af instruktion.
Instruktion dekodes. Hvilken type, og hvilken operand, der behøves.

S3: Fetch af operand.
Finder operander fra register eller hukommelse.

S4: Eksekvering af instruktion.
Data køres igennem data-path og instruktionen eksekveres.

S5: Skriv tilbage.
Gem i register.

Mic-3 har kun fire faser; intet S2.

Seriel CPU:

Instruktion a					Instruktion b					Instruktion c				
S1	S2	S3	S4	S5	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Hvis tre instruktioner eksekveres sekventielt, vil det tage $15\,\Delta T$, hvor en clock-cyklus tager ΔT.

Pipelined CPU:

Instruktionsfetch	S1:	a	b	c	d	e	f	g
Instruktionsdekodning	S2:		a	b	c	d	e	f
Operand-fetch	S3:			a	b	c	d	e
Eksekvering	S4:				a	b	c	d
Gem i register	S5:					a	b	c
		1	2	3	4	5	6	7

De samme tre instruktioner kan udføres på $7\,\Delta T$, hvis de udføres i en pipeline.

Parallelisme på instruktionsniveau.

En pipeline giver anledning til en afvejning mellem forsinkelse (latency) og båndbredde (bandwidth).

Med $T = 2\,\text{ns}$ per fase, og der er $n = 5$ faser, tager én instruktion:

$$T \cdot n = 2\,\text{ns} \cdot 5 = 10\,\text{ns},$$

hvilket er forsinkelsen (latency).

En pipeline giver anledning til en afvejning mellem forsinkelse (latency) og båndbredde (bandwidth).

Med $T = 2\,\text{ns}$ per fase, og der er $n = 5$ faser, tager én instruktion:

$$T \cdot n = 2\,\text{ns} \cdot 5 = 10\,\text{ns},$$

hvilket er forsinkelsen (latency).

Hver fase tager 2 ns, dvs. en instruktion er færdig hvert andet ns.
Altså er CPUens båndbredde:

$$(2\,\text{ns})^{-1} = \frac{1}{2} 10 \times 10^9\,\text{IPS} = 500\,\text{MIPS},$$

hvor IPS er enheden *Instruktioner Per Sekund*.

Tre typer af pipeline-risici:

- Strukturelle risici:
  Opstår ved resource-konflikter.
  Når hardware ikke understøtter bestemte kombinationer af instruktioner samtidigt.
  Kan være nødvendigt at lave en pipeline-stall.
- Data-risici:
  Opstår når en instruktion afhænger af resultatet fra en tidligere instruktion, hvis dette ikke eksisterer pga. instruktions-overlap.
- Kontrol-risici:
  Opstår ved forgreninger der ændrer pipelinen.

Ressourcekonflikt.

I hver fase af pipelinen benyttes forskellige ressourcer, f.eks.:

Pipeline-fase	Anvendte ressourcer
Fetch af instruktion	PC, MAR, adresse- og databus
Decode	Decoder, intern bus, MAR, PC, adresse- og databus
Fetch af operand	PC, MAR, adresse- og databus
Eksekvering	ALU, intern bus
Skriv tilbage	Intern bus, MAR, adresse- og databus

Gengangere skaber problemer.

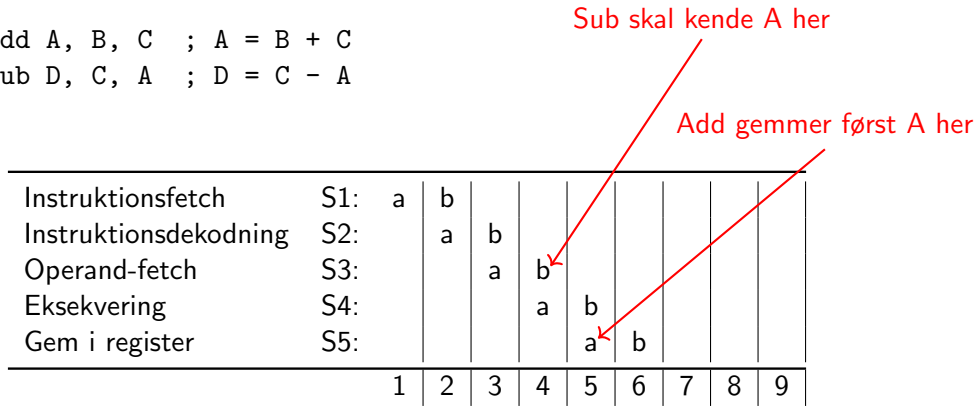
Gengangere skaber problemer.

Løsning:
Duplikering af ressourcer kan afhjælpe problemet:

- Dual-cache for at dele kode og data op med separate busser og MAR.
- Multi-port CPU-registre og duplikering af interne busser.

Betragt koden:

```
a: add A, B, C  ; A = B + C
b: sub D, C, A  ; D = C - A
```



Ved at indsætte et vent i to cyklusser, kan problemet løses.

Instruktionsfetch	S1:	a	-	-	b					
Instruktionsdekodning	S2:		a	-	-	b				
Operand-fetch	S3:			a	-	-	b			
Eksekvering	S4:				a	-	-	b		
Gem i register	S5:					a	-	-	b	
		1	2	3	4	5	6	7	8	9

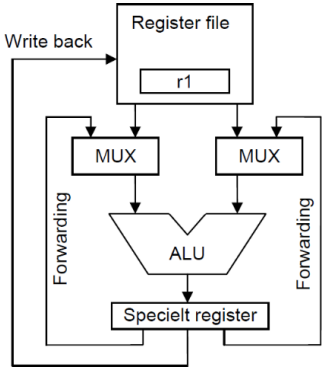
Problemer:

- Formindskelse af throughput og ydeevne.

- Out-of-order execution
  'Hullet' efterladt efter et pipeline-stall kan fyldes ud af andre processer, der ikke har noget at gøre med process a og b.

Instruktionsfetch	S1:	a	c	d	b					
Instruktionsdekodning	S2:		a	c	d	b				
Operand-fetch	S3:			a	c	d	b			
Eksekvering	S4:				a	c	d	b		
Gem i register	S5:					a	c	d	b	
		1	2	3	4	5	6	7	8	9

- Forwarding
  Brug af et specielt register, der gemmer resultatet i eksekverings-skridtet, så det kan bruges i en anden beregning.
  Resultat gemmes stadig i registret.



Forgrenings-hazard.
Hvis en instruktion viser sig at være f.eks. en `goto`:

- Tøm pipelinen; alle tidligere hentede instruktioner er ugyldige.
- Hent instruktioner fra den nye lokation.
- Lavere ydeevne.

Forgrenings-hazard.
Hvis en instruktion viser sig at være f.eks. en `goto`:

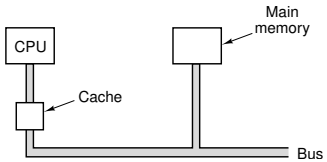
- Tøm pipelinen; alle tidligere hentede instruktioner er ugyldige.
- Hent instruktioner fra den nye lokation.
- Lavere ydeevne.

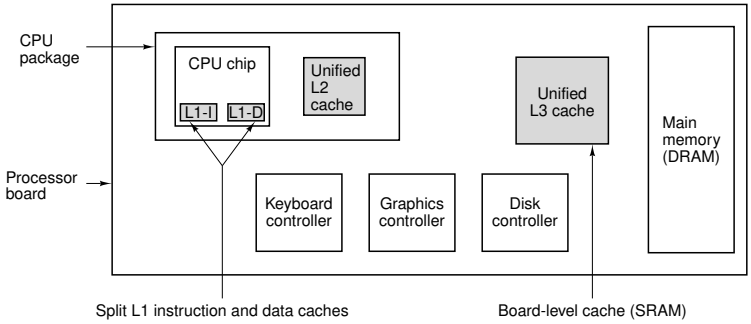
Ved en betinget forgrening ved man først, hvor man skal hen, når den er eksekveret.
Mulige løsninger:

- Forgreningsforudsigelse:
  Lad CPUen forsøge at gætte resultatet af en branch.
  Gør brug af kendskab; f.eks. vil en forgrening baglæns ofte blive taget fremfor en forgrening fremad. Tænk på while-, for-loop, etc.
- Forgrenings-cache:
  Gem de første instruktioner i de to forskellige forgreninger i en speciel cache, så de hurtigt kan loades ind i pipelinen.

Indhold

- Den primære hukommelse er væsentligere langsommere end CPUen.
- Når CPUen beder om data fra hukommelsen, kan den være nødt til at vente i mange cyklusser.
- Løsning: Placer hurtig hukommelse direkte på CPU-chippen.
  - En stor hukommelse gør chippen større og derfor dyrere.
- Vi vil gerne have en stor mænge meget hurtig hukommelse.
- En kombination af en meget hurtig cache på CPU-chippen og en langsom primær hukommelse.





- Flere cache forskellige steder i computeren.
- L1 $\subset$ L2 $\subset$ L3 $\subset$ primær hukommelse.

Cache

Baggrund

Idé:

- De mest brugte words fra hukommelsen gemmes i hukommelsen.
- Når CPUen skal hente et word, kigger den først i cachen.
- Kun hvis det ikke er at finde der, kigger den i hukommelsen.

Cache

Baggrund

Idé:

- De mest brugte words fra hukommelsen gemmes i hukommelsen.
- Når CPUen skal hente et word, kigger den først i cachen.
- Kun hvis det ikke er at finde der, kigger den i hukommelsen.

Lokalitetsprincippet:

- Et program tilgår ikke hukommelsen totalt tilfældigt.
- Den næste operation eller det næste data er ofte i nærheden af den nuværende.
- Når et word tilgås vil det og nogle naboer flyttes fra den langsomme hukommelse til den hurtige cache.
- To typer:
  - **Spatial locality**: Adresser med numerisk lighed har stor sandsynlighed for at blive tilgået.
  - **Temporal locality**: Stor sandsynlighed for at tilgå adresse, der lige har været tilgået.

Tilgangstid

Tilgangstiden kan beregnes vha.

$$\text{tilgangstid} = c + (1 - h)m,$$

hvor c er tiden til at hente fra cache, m er tid, det tager at hente fra hukommelse, h er hit-ratio.

Tilgangstid

Tilgangstiden kan beregnes vha.

$$\text{tilgangstid} = c + (1 - h)m,$$

hvor c er tiden til at hente fra cache, m er tid, det tager at hente fra hukommelse, h er hit-ratio.

Eksempel:

$$\begin{aligned} c &= 5\,\text{ns} \\ m &= 60\,\text{ns} \\ h &= 0.9 \\ &\Rightarrow \text{tilgangstid} = 5\,\text{ns} + (1 - 0.9) \cdot 60\,\text{ns} = 11\,\text{ns} \end{aligned}$$

Opdeling i Cache-linjer

- Den primære hukommelse er delt op i blokke med fast størrelse kaldet cache-linjer (cache lines).
- Størrelsen er typisk 4 til 64 bytes.
- Linjerne er nummereret efter hinanden, dvs. for 64 byte linje-størrelse:
  - Linje 0 er byte 0 til 63
  - Linje 1 er byte 64 til 127
  - osv.
- Cache-controlleren checker om hukommelse ligger i cachen eller i primær hukommelse og handler derefter.

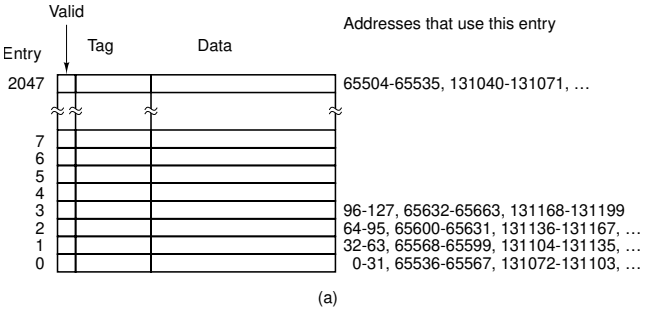
- Den simpleste cache.
- Hver linje (entry) kan holde én cache-linje fra hukommelsen.
- Eksempel:
  - 32-byte cache-linjer.
  - 2048 entries $\Rightarrow 2048 \cdot 32\,\text{bytes} = 64\,\text{KB}$

Entry:

- **Valid** bit
- 16-bit **Tag**
- 32-byte **Data**

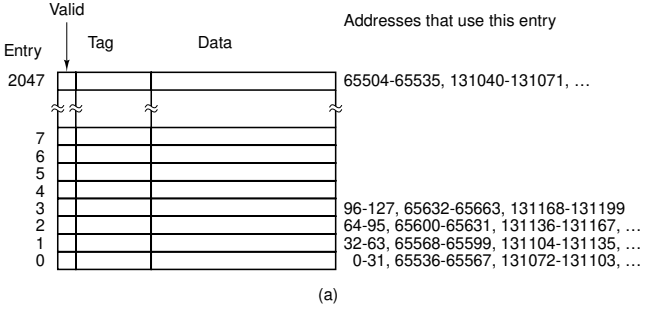
Hukommelses-adresse:

- **TAG** tilsvarer **Tag** i entry.
- **LINE** indikerer hvilken entry, der holder data.
- **WORD** indikerer hvilket ord, der refereres til.
- **BYTE** indikerer hvilket byte, der refereres til.



Problem:

- Adresser med et helt antal 2^{16} imellem sig vil ligge i samme entry pga. samme værdi af **LINE**.
- Vil føre til kollision, der gør ydeevne meget lav.
- Kan løses af compiler.



Problem med direct-mapped:

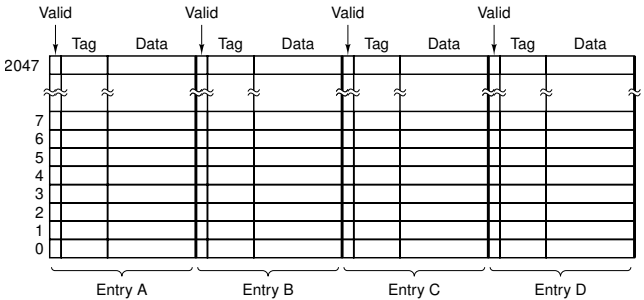
- Mange adresser vil mappe til den samme adresse.
- Giver anledning til konflikter.

Løsning:

- To eller flere linjer i hver entry.

Indsættelse af ny data. Hvilken skal man fjerne?

- Erstat den data, der blev tilgået for længst tid siden.
- Least Recently Used (LRU).



Figur: Four-way set-associative cache.

Referencer I

[1] A. S. Tanenbaum, T. Austin og B. Chandavarkar, **Structured computer organization**, eng, 6. edition. International edition. Boston, Mass: Pearson, 2013.

Opgave

- Oversæt følgende instruktioner fra Mic-2 til den pipelinede Mic-3 og sammenlign antal cyklusser:
  - `IADD`
  - `ISTORE`
  - `ILOAD`
  - `POP`
  - `INVOKEVIRTUAL`
- Løs opgave 4.32 i [1], s. 341. Benyt dit foretrukne sprog, f.eks. Python, C++, etc.