

UiO : Department of Informatics
University of Oslo

Forelesning 7b IN2060

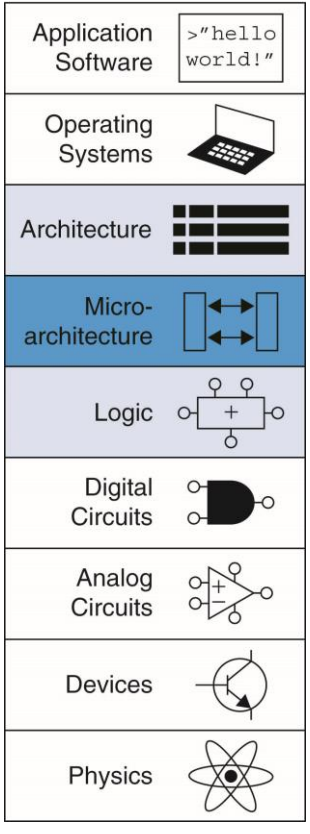
Mikroarkitektur 2

Pipelining m.m.



Mikroarkitektur - Pipelining++

- Repetisjon av single-cycle og multicycle
- Pipelining
  - Forenklet 5 stegs versjon
  - Hazards, stall, forwarding, Early BTA
- Se på noen avanserte metoder
- Litt ARM historie
- Hva er eksamens relevant?
  - Forståelse og foilene



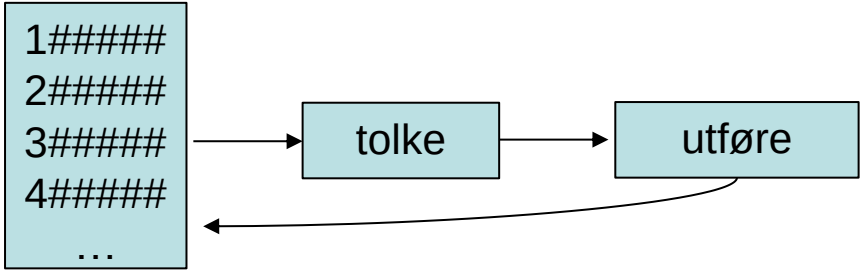
Implementasjon av mikroarkitektur

Husk! - Vi har lært det vi trenger for å implementere en arkitektur

- Kombinatorisk og synkron logikk.
- Verktøy: Tilstandsmaskiner og digitale byggeblokker
- Spesifisert arkitekturen (ARM), som er definert av
  - Instruksjonene som må støttes
  - Arkitektur tilstandene (R0-R15 og *status*-registeret)

Hvordan implementere?

- *Mikroarkitekturen*: En spesifikk implementasjon/løsning som utfører instruksjonene, slik arkitekturen krever.



- Det er opp til mikroarkitekturen å finne ut hvordan dette skal løses.

Ytelse ?

- Løsningen - et kompromiss mellom
  - Kostnad
  - Strømforbruk
  - Hastighet (hvor raskt den faktisk jobber)
- Klokkefrekvensen bestemmer ikke ytelsen alene
  - En detaljert analyse er nødvendig.
- MIPS – (Million Instructions Per Second)
  - Nyttig hvis man sammenligner like arkitekturer
- *Benchmarks* = Ytelsestest.
  - Programmer som måler spesifikke ytelser (f.eks. heltall, flyttall).
  - Må velge den som er mest relevant.

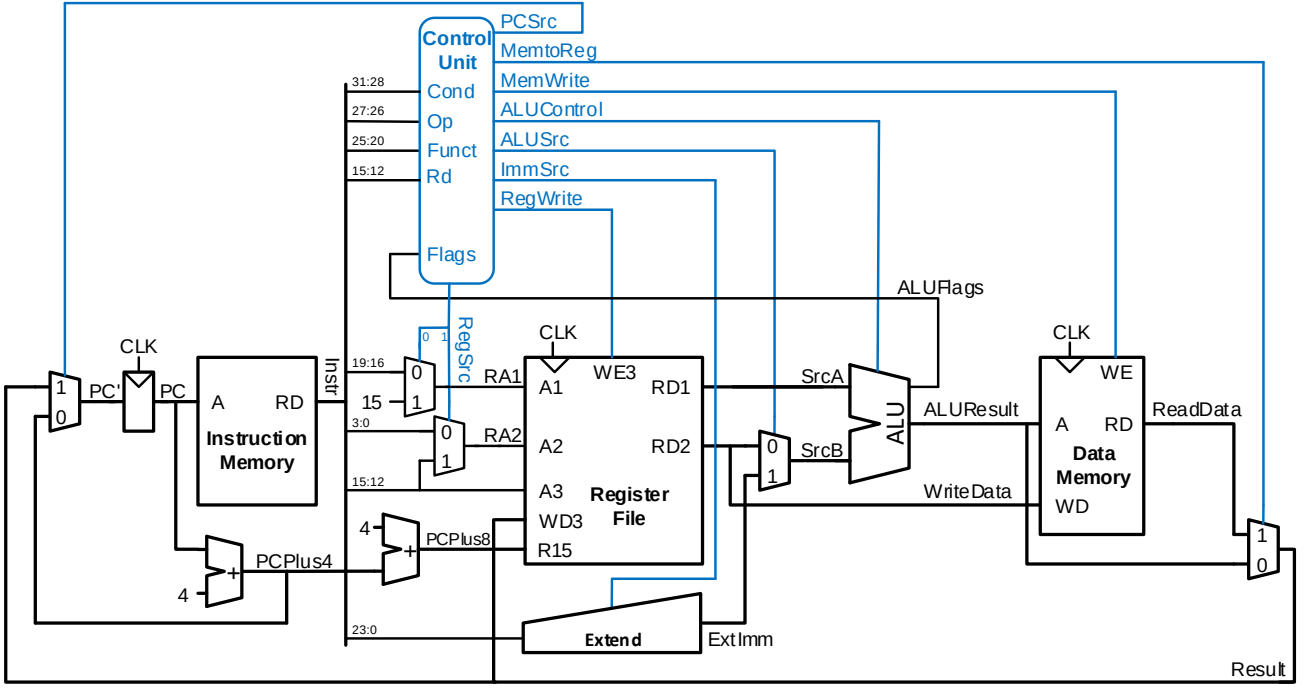
Single-cycle

Deler mellom

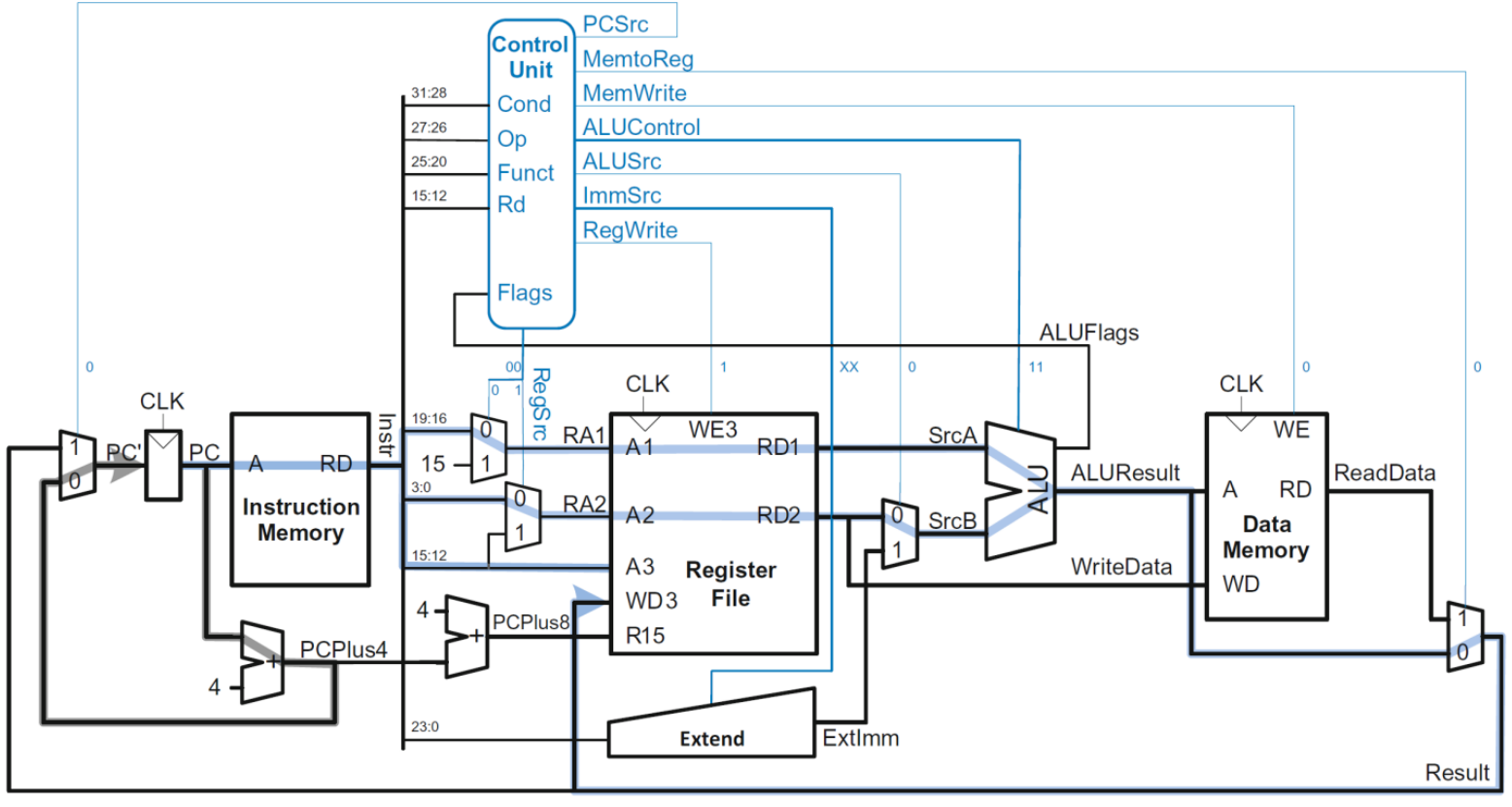
- Dataflyten
- Kontroll (blå)

Instruksjonen blir utført i løpet av en klokkeperiode. Dvs. CPI = 1.

Kontroll logikken er derfor relativ enkel. (Kombinatorisk + ALUFlags register)

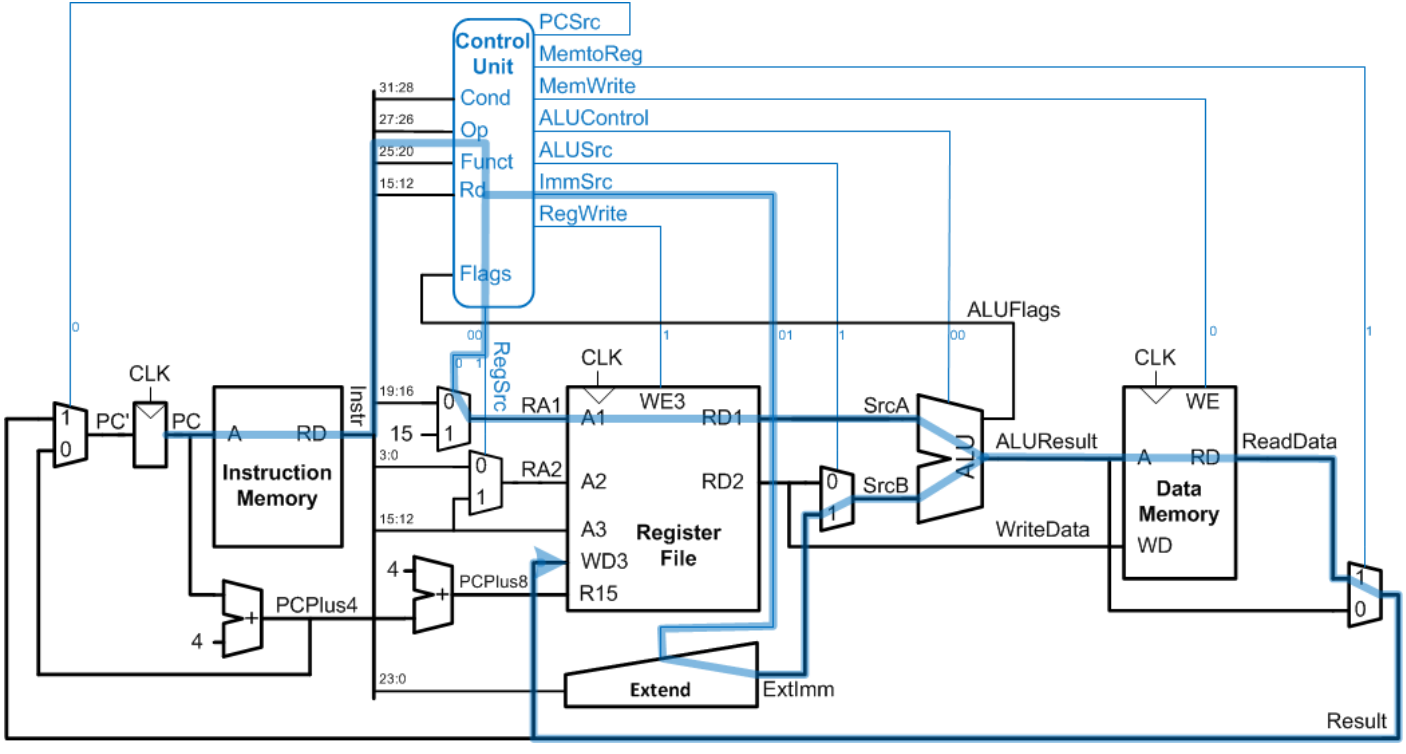


Eksempel: `ORR Rd, Rn, Rm`



Vi må finne den klokkehastigheten som fungerer for alle instruksjonsene

Single Cycle: Maks klokkehastighet



T_C er begrenset av *critical path* (`LDR`)

Multi-cycle

Vi deler opp instruksjonene i mindre deler.

Fordeler:

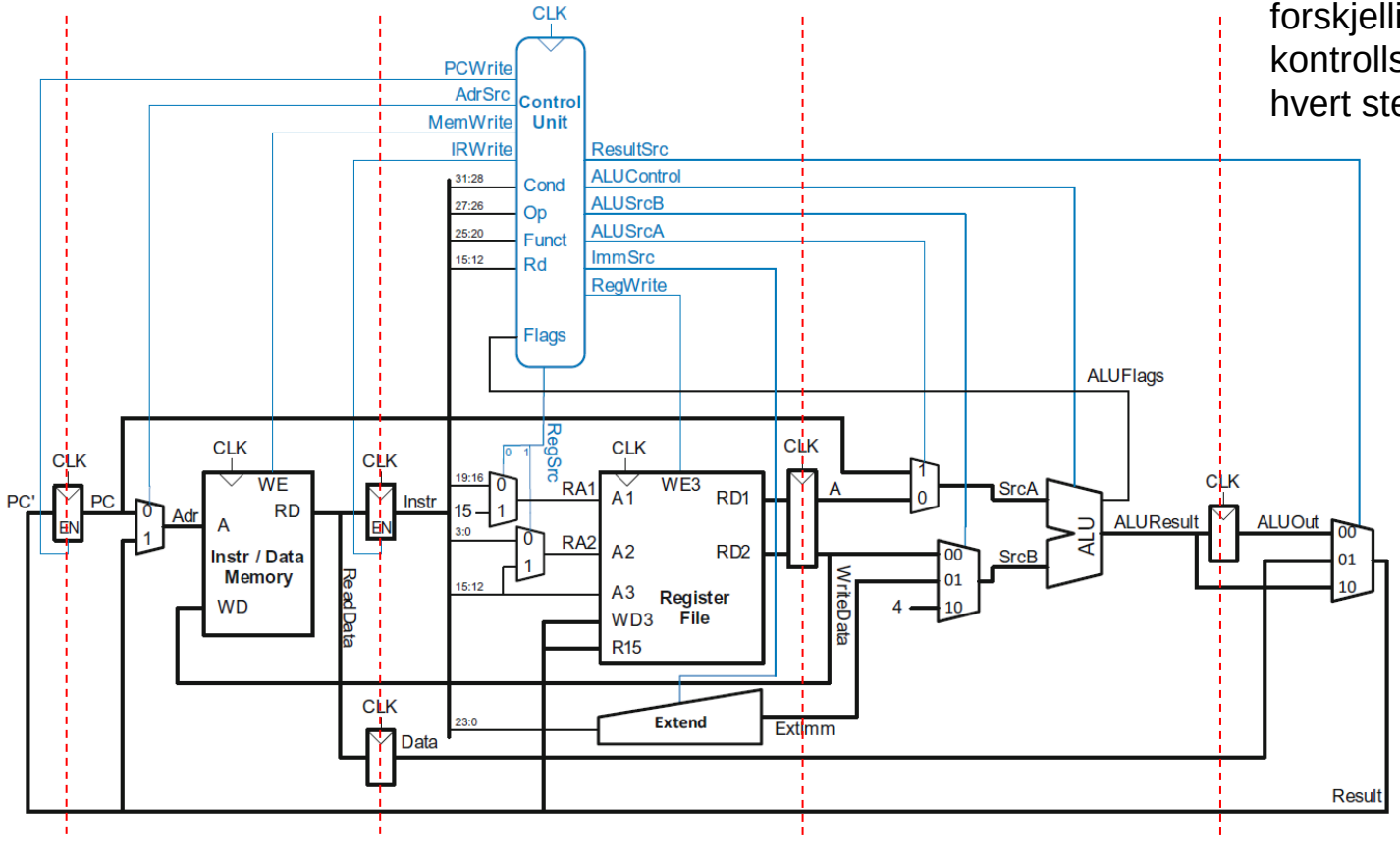
- Vi kan ha en høyere klokkehastighet..
- De enkleste instruksjonene utføres raskere
  - **B** tar 3 steg, mot **LDR**'s 5.
- Vi kan gjenbruke dyr/plasskrevende hardware

Ulemper:

- Vi får overhead pga. «buffring» mellom stegene

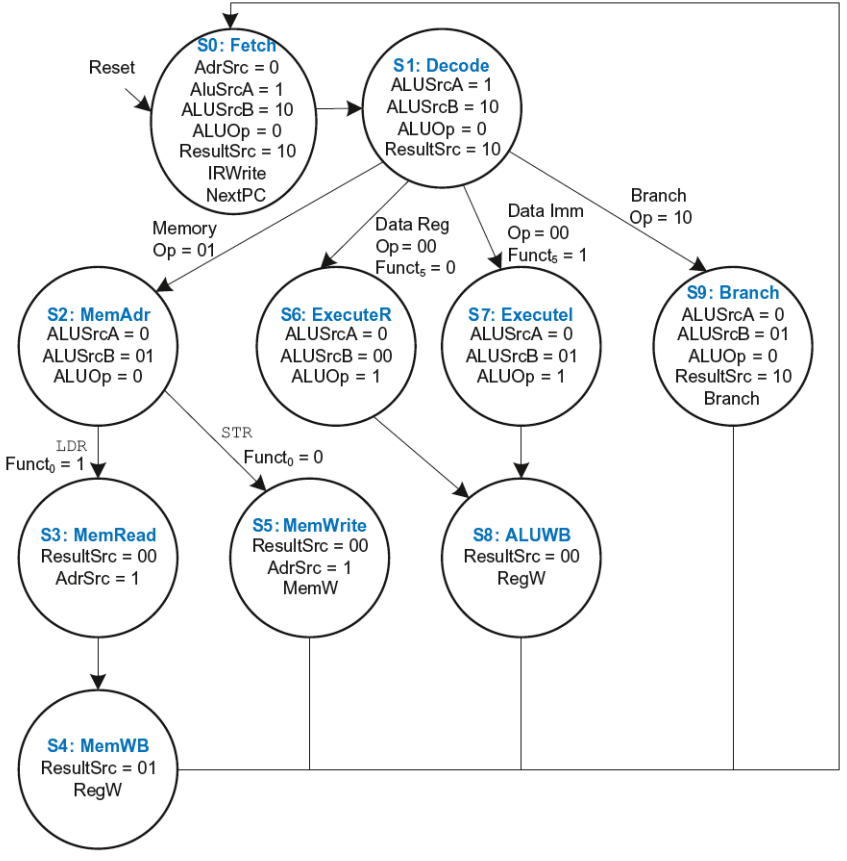
Multicycle – deler opp med register

Merk: Hver instruksjon trenger forskjellige kontrollsignaler for hvert steg



Multicycle kontroll = Tilstandsmaskin

- Siden en instruksjon varer over flere klokkesykler, må vi har forskjellige kontrollsignaler for hvert steg av instruksjonen.
- Løser dette med en tilstandsmaskin



Multicycle ytelse

- Får høyere klokkehastighet enn *Single-Cycle*, færre porter å propagere. Men hver instruksjon består nå av flere cycler.
- Siden instruksjonen har forskjellige lengder, må vi finne en gjennomsnittlig CPI (vha. statistikk).
- Kan bruke Benchmarks til dette.
  - SPECINT2000: CPI = 4.12.
- 100 billioner instruksjoner.
  - 140 sekunder (vs. Single-Cycle's 84 sekunder)
  - Tregere grunnet overhead og høy CPI.

Pipeline

Pipelining – *temporal parallellisme*

- Tilsvarende oppdeling av hardware/instruksjoner som multicycle.
- Men vi starter nye instruksjoner med engang etter at forrige instruksjon er startet
- Vi får bedre utnyttelse av prosessoren/hardware
  - Dvs. vi bruker hver del av prosessoren mer effektivt.

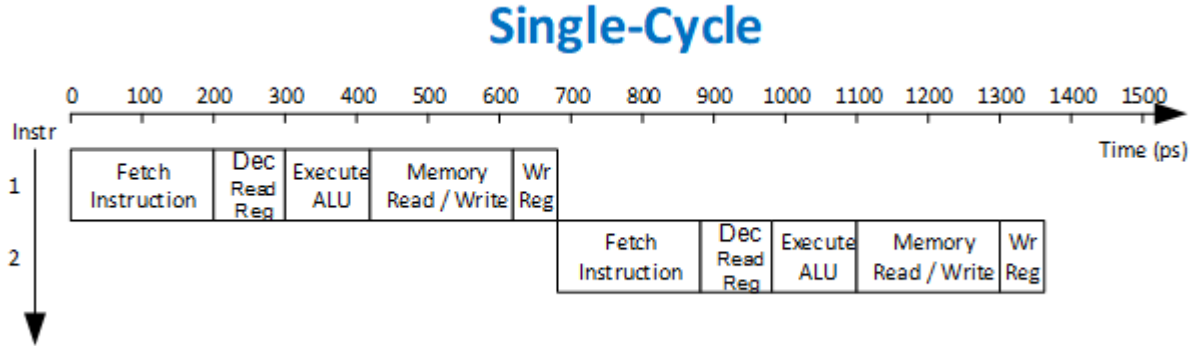
Pipelined ARM Processor

- Deler single-cycle processor inn i 5 deler:
  - Fetch
  - Decode
  - Execute
  - Memory (w/r)
  - Writeback (to RF)
- Legger *pipeline buffer'*e mellom hver del
- *Starter ny instruksjon før den forrige er ferdig!*
  - *Temporal parallelism*

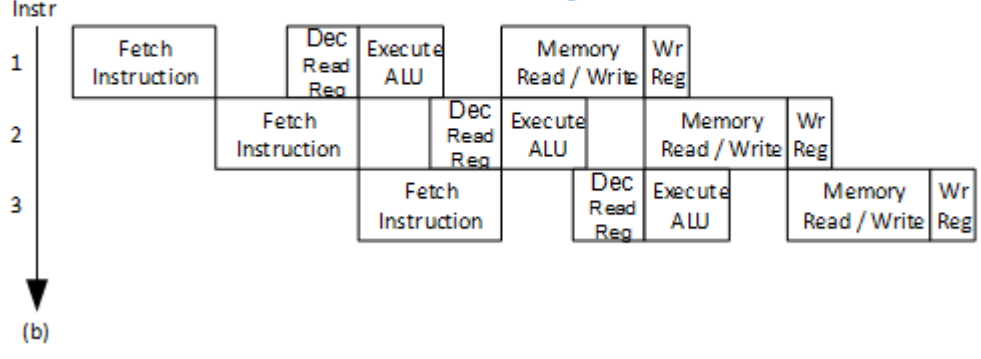
Merk: alle instruksjoner må «passe inn» i denne oppdelingen

Merk: En single-cycle instruksjon er raskere enn en pipeline instruksjon grunnet overhead.
Men 3 pipeline instruksjoner er raskere enn 2 single-cycle-instruksjoner.
Hvor mye raskere er pipeline?

Single-Cycle vs. Pipelined



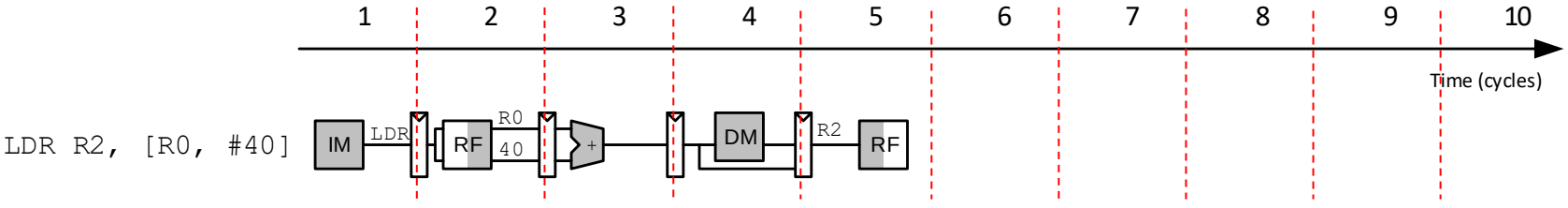
- Må vente til forrige instruksjon er ferdig før den kan starte ny.



- Starter kontinuerlig nye instruksjoner.

Merk at ikke alle instruksjoner trenger steg 4 og 5 (grå/hvit)

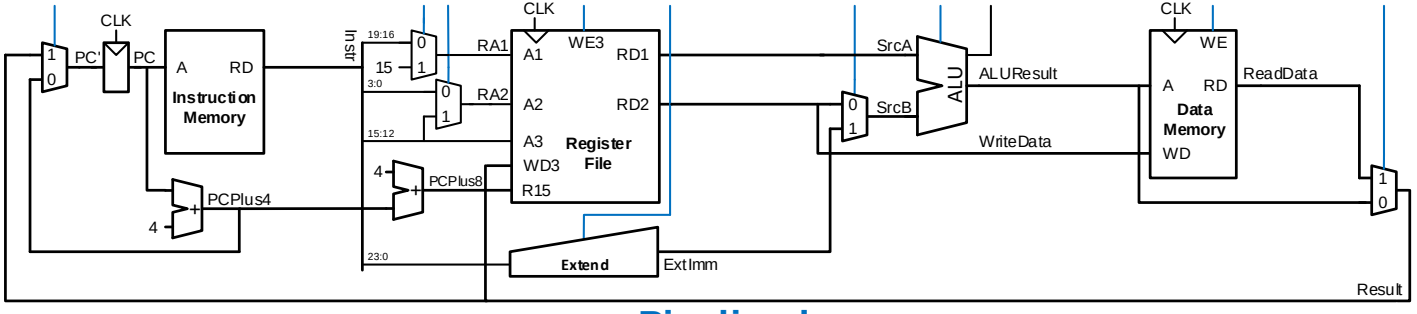
Pipelined Processor Abstraction



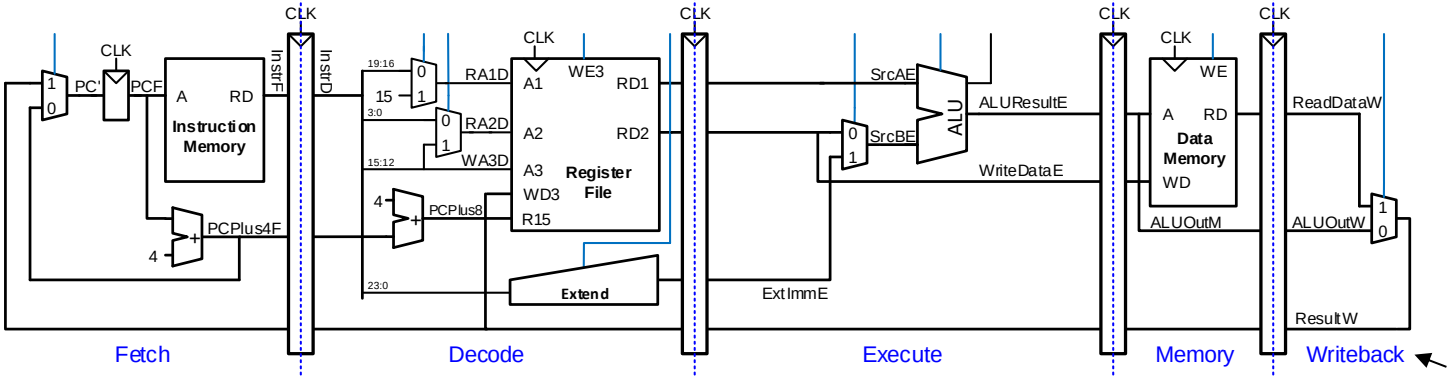
Merk RF W/R i samme *cycle* (leser/skriver på positiv/negative klokkeflanke)

Single-Cycle & Pipelined Datapath

Single-Cycle



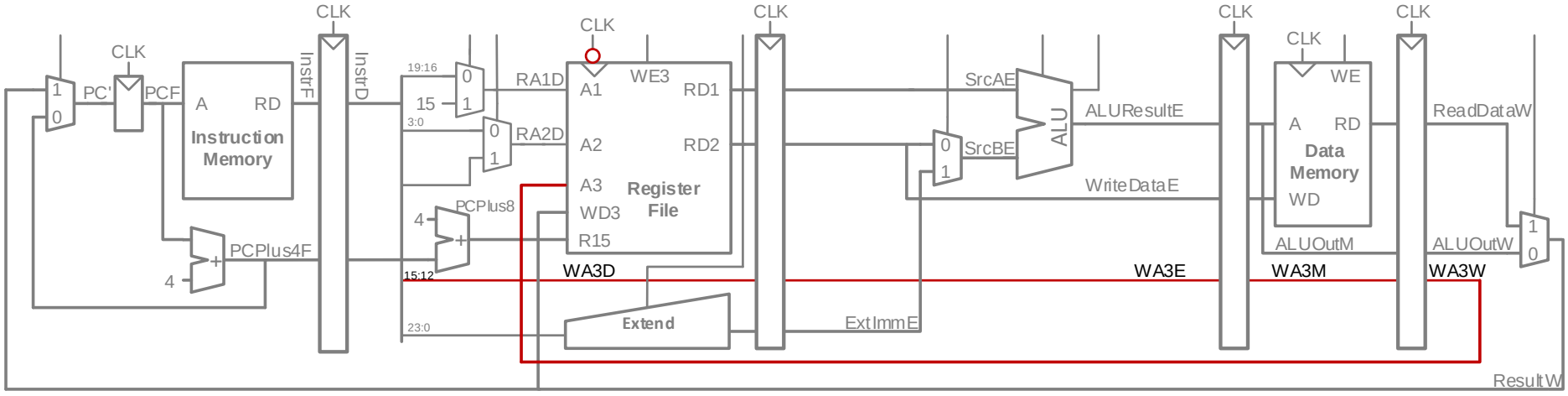
Pipelined



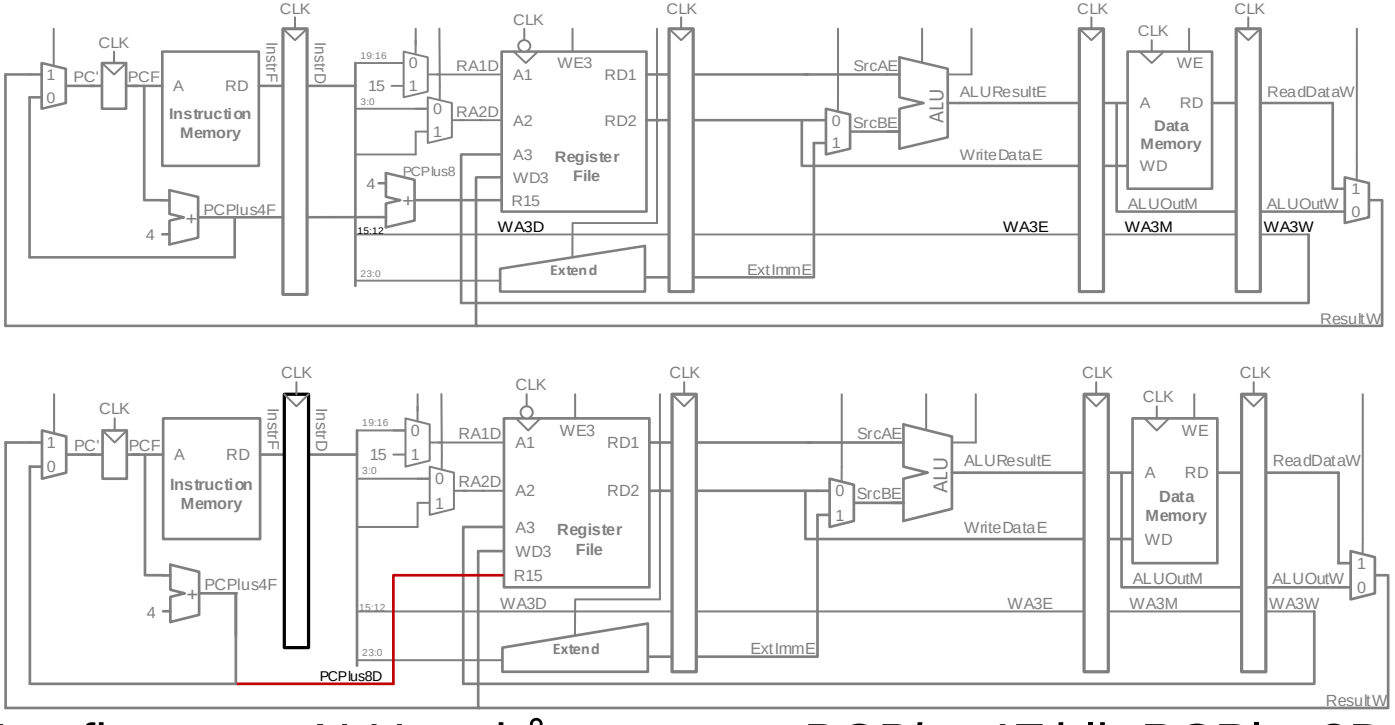
Merk navnene på inndelingen

Corrected Pipelined Datapath

- *WA3* register adresse må ankomme RF samtidig som det resultatet vi ønsker å skrive til RF (*ResultW)* F.eks. `ADD R0 R1 R2`.
- *Løser dette med å forsinke WA3D adressen med Pipeline buffrene.*
  - *Sørger for at WA3W og ResultW hører sammen.*



Optimized Pipelined Datapath

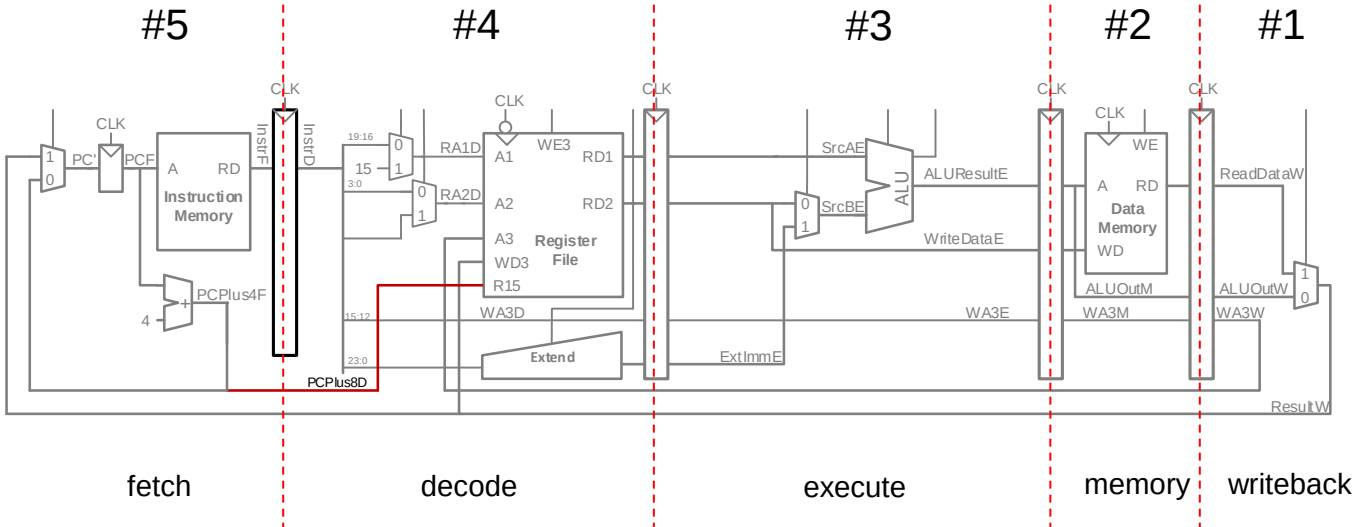


- Kan fjerne en ALU ved å utnytte at *PCPlus4F* blir PCPlus8D i decode fasen.

Pipeline eksempel, PCPlus4F = PCPlus8D ?

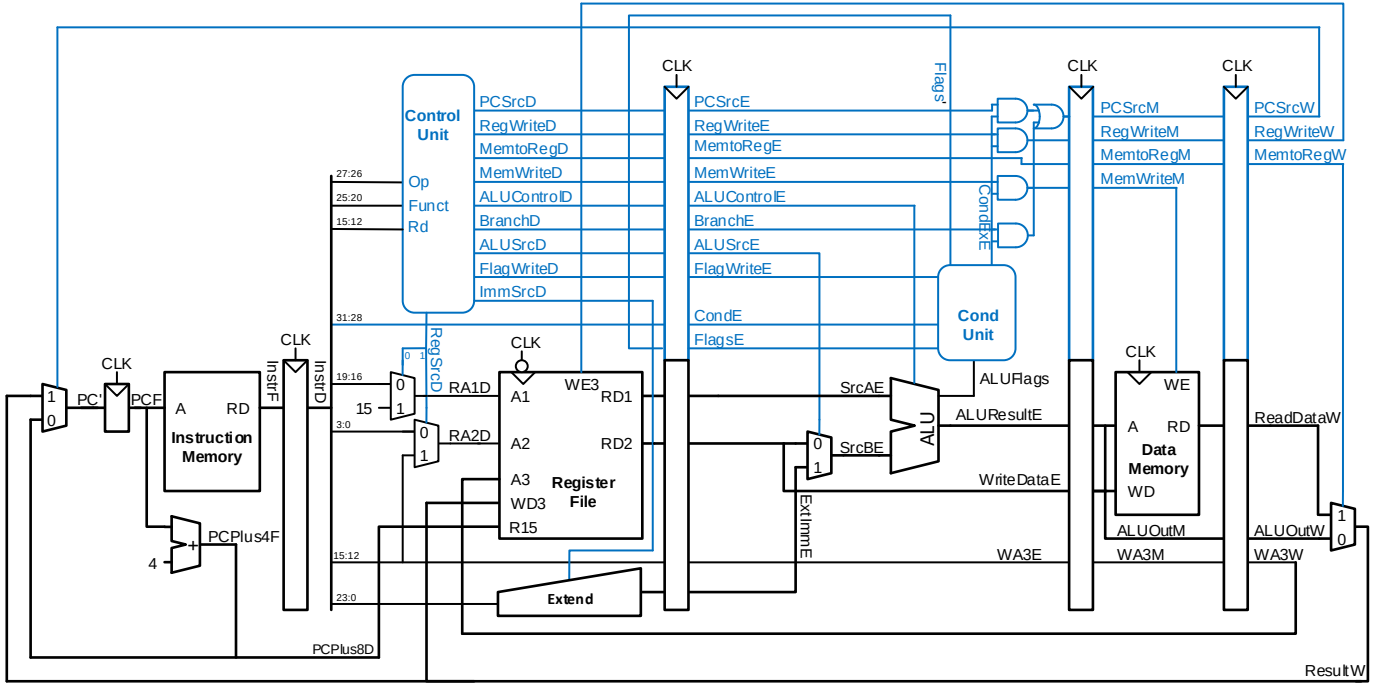
#1 0xEXXXXXX
#2 0xEXXXXXX
#3 0xEXXXXXX
#4 0xEXXXXXX
#5 0xEXXXXXX
#6 0xEXXXXXX
…

- Hvilke instruksjoner vil de forskjellige delene jobbe med etter 5 klokkesykler?
  - *Fetch* starter med instruksjon #5 mens writeback avslutter instruksjon #1
- Merk at *fetch* regner ut neste instruksjon PC' = PCF+4 = #5 + 4 = #6
- Samtidig trenger *decode* å regne ut PCD + 8 = #4 + 8 = #6!
  - Mao. blir fetch sin PCPlus4F, det samme som PCPlus8D for decode.
  - (Branch gir et potensielt problem, se fotnote s. 429)



Utvider pipline buffrene til å også omfatte kontrollsignalene. De tolkede kontrollsignalene *følger instruksjonens gang* i pipelinen.

Pipelined Processor Control

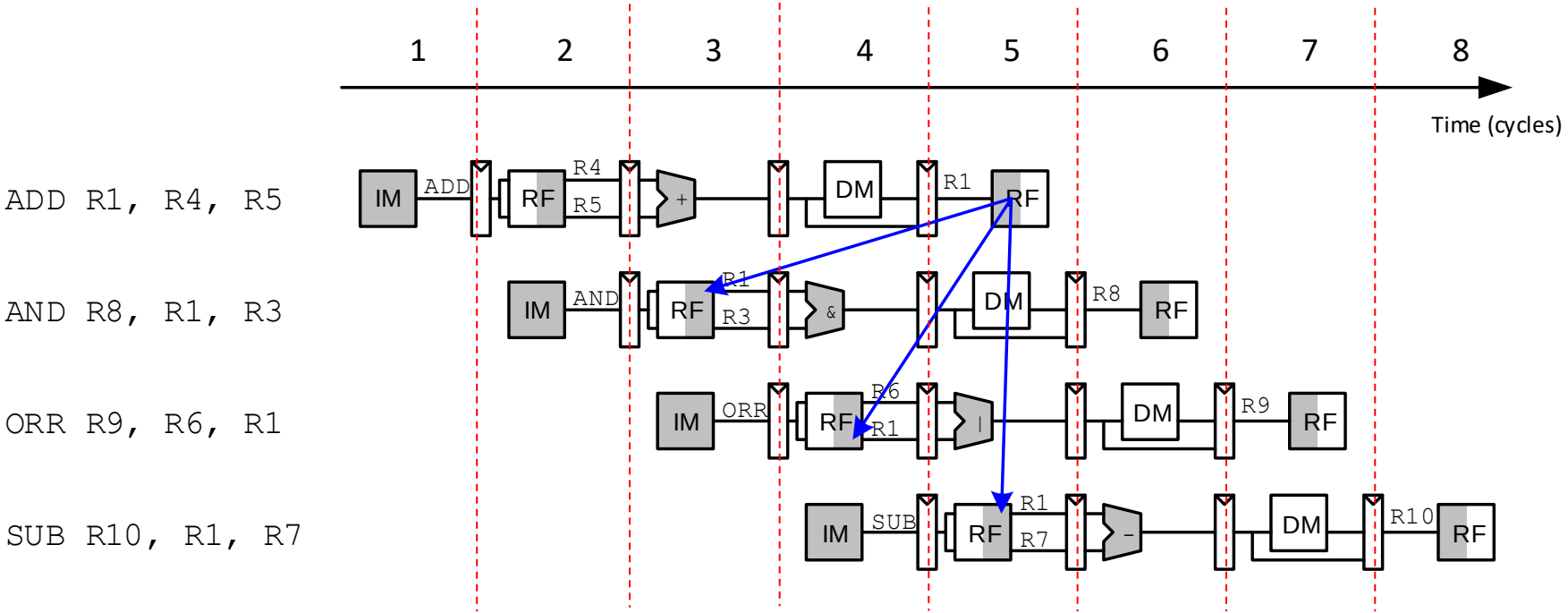


- Tilsvarende *control unit* som single-cycle processor
- Kontrollsignalene blir korrekt forsinket vha. pipeline buffrene.

Pipeline Hazards

- Når en instruksjon er avhengig av resultat fra en instruksjon som ikke er ferdig enda.
  - Dette problemet har vi ikke med single-cycle
- To typer:
  - **Data hazard:** verdi er enda ikke skrevet til register-filen
    - Dvs. data er ikke klar når vi trenger det.
  - **Control hazard:** Neste instruksjon er ikke avgjort enda
    - F.eks. cmp ikke ferdig.

Data Hazard



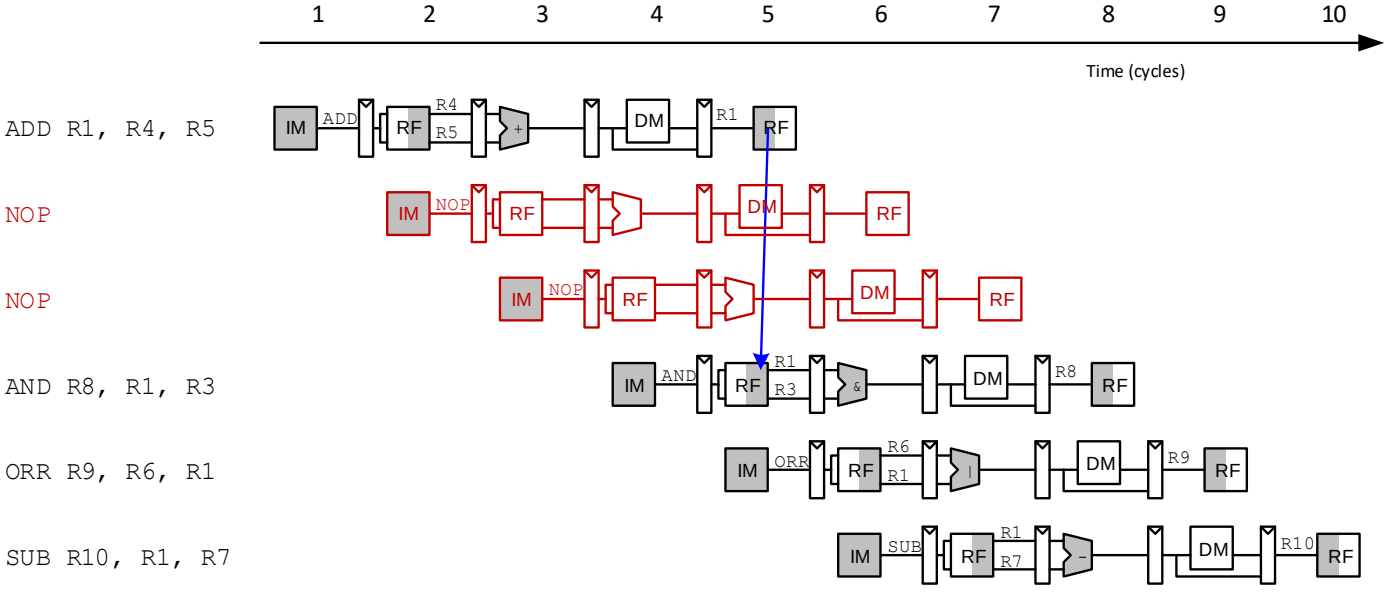
- R1 er ikke ferdig før i klokkeperiode 5
- SUB instruksjon går fint siden vi skriver/leser på neg/pos klokkeflanke (merk aktivitets/gråfarge forskjellen)

Håndtere *data hazards*

- Legg inn `NOP`s i koden under kompileringen
  - NOP – *no operation*
- Omorganisere koden under kompileringen
- *Forward* data under kjøringen
  - Gjør data tilgjengelig når den er ferdig utregnet
- *Stall* prosessoren under kjøring (sette på vent)

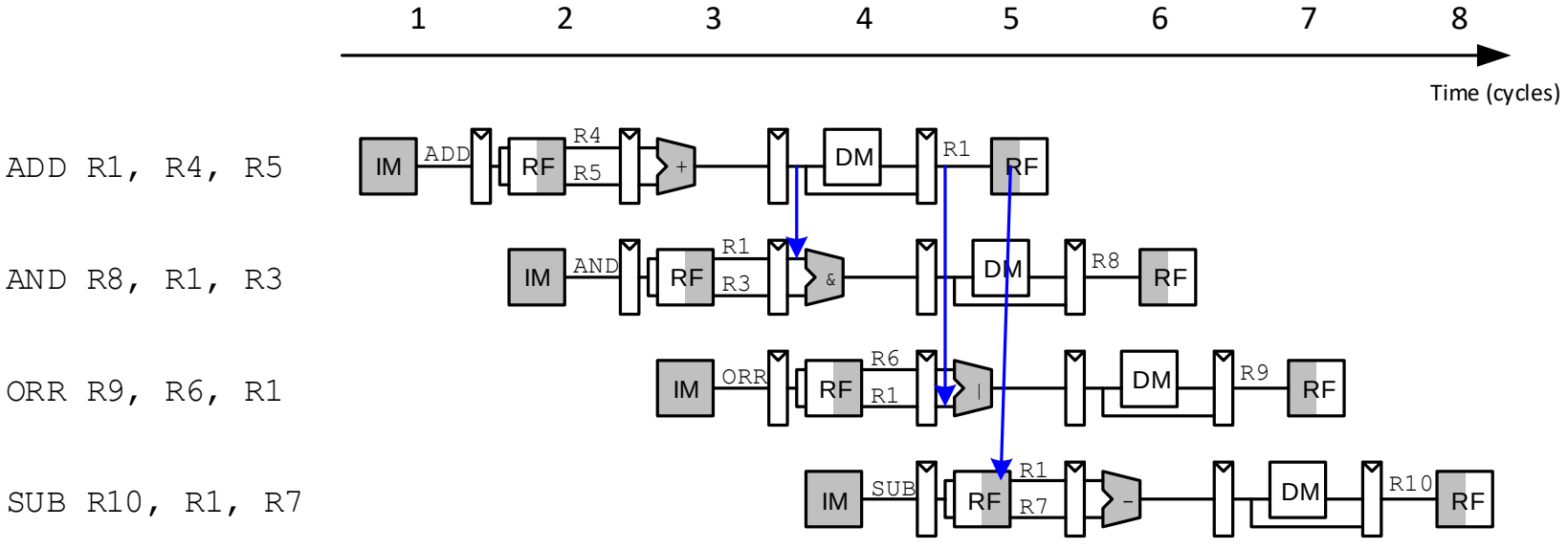
Håndtere *Hazard* under kompilering

- Legg inn `NOP`s slik at resultatet blir ferdig i tide
- Eventuelt, legg inn «uavhengige» funksjoner (omorganisere)



Muligheten for å hente ut dataene direkte fra datapathen

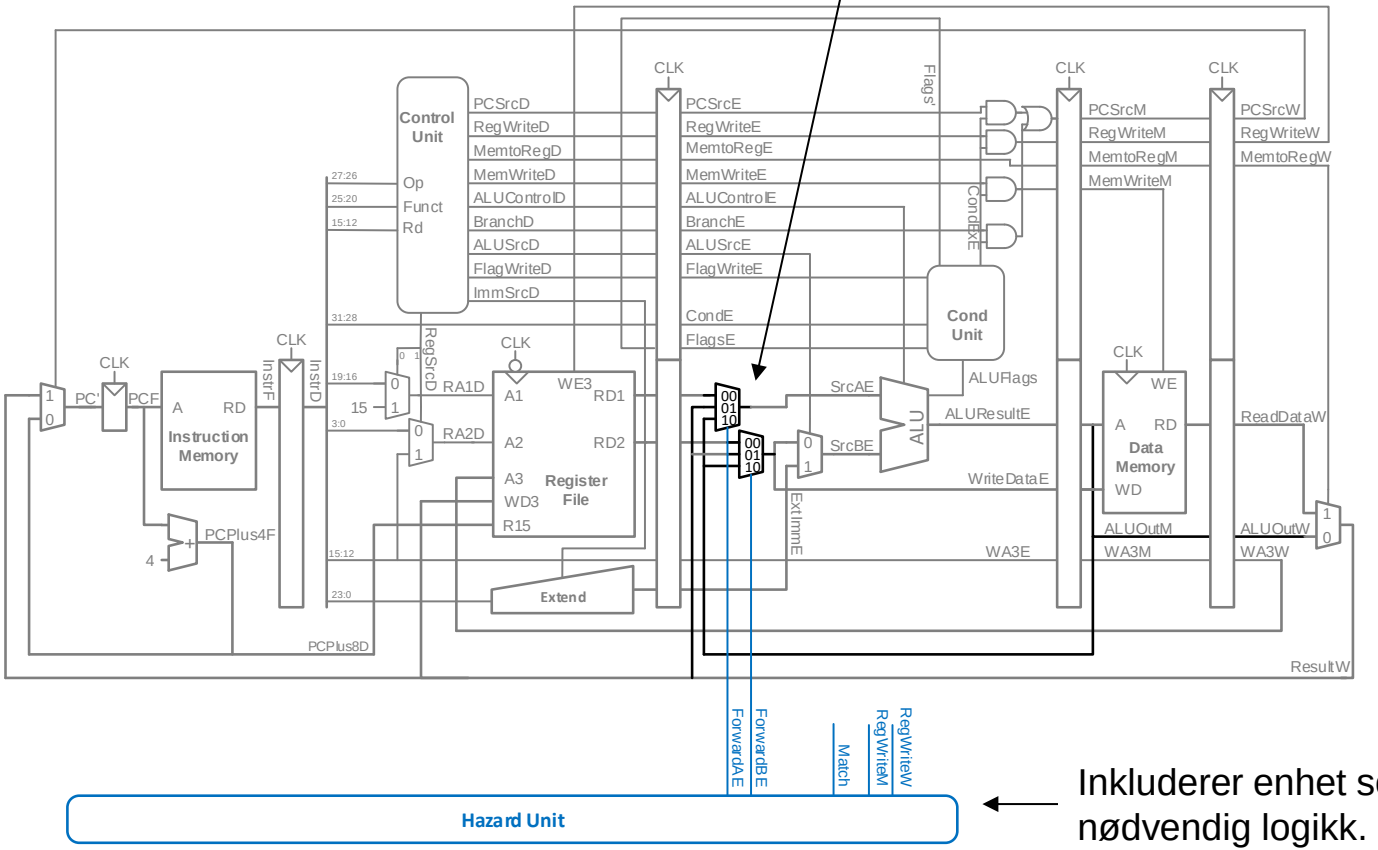
Data forwarding



- Sjekk om *register read* i *Execute* delen passer med *Register write* i memory eller *writeback stage*
  - I så fall kan vi “framskynde” resultatet

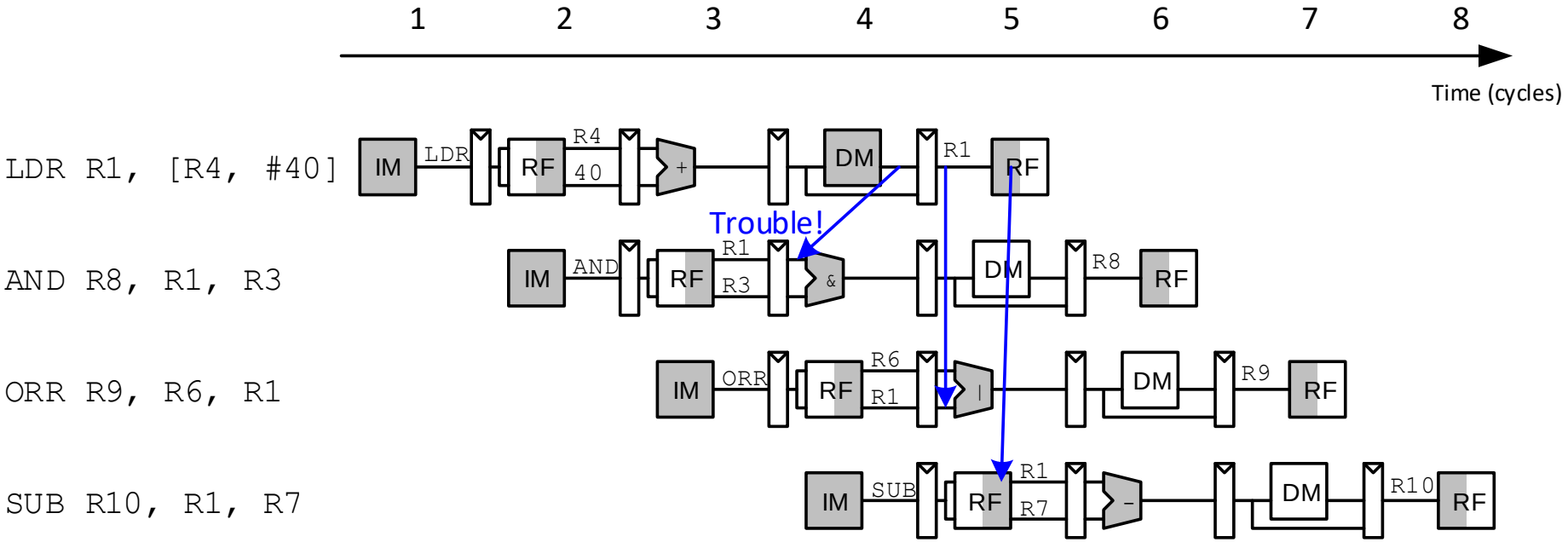
Data forwarding

MUX'ene sørger for at ALU også har tilgang til data fra datapathen.



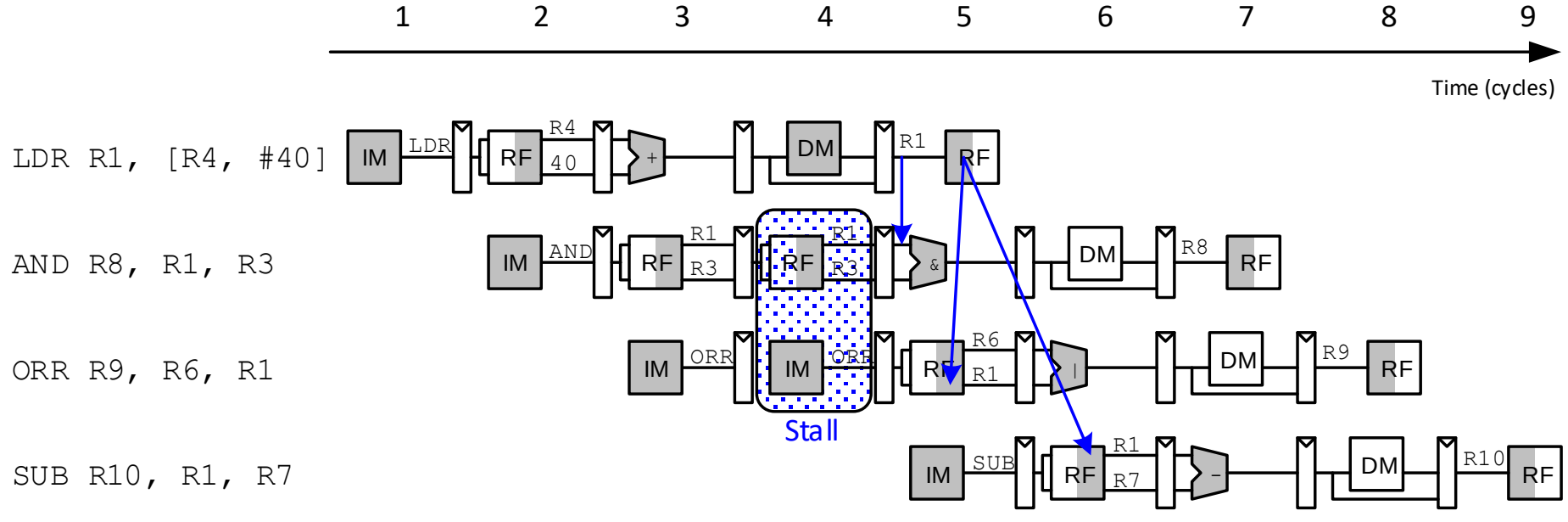
Inkluderer enhet som tar seg av nødvendig logikk.

Stalling



Verdien R1 = *mem(R4+#40) er ikke klar før steg 5*

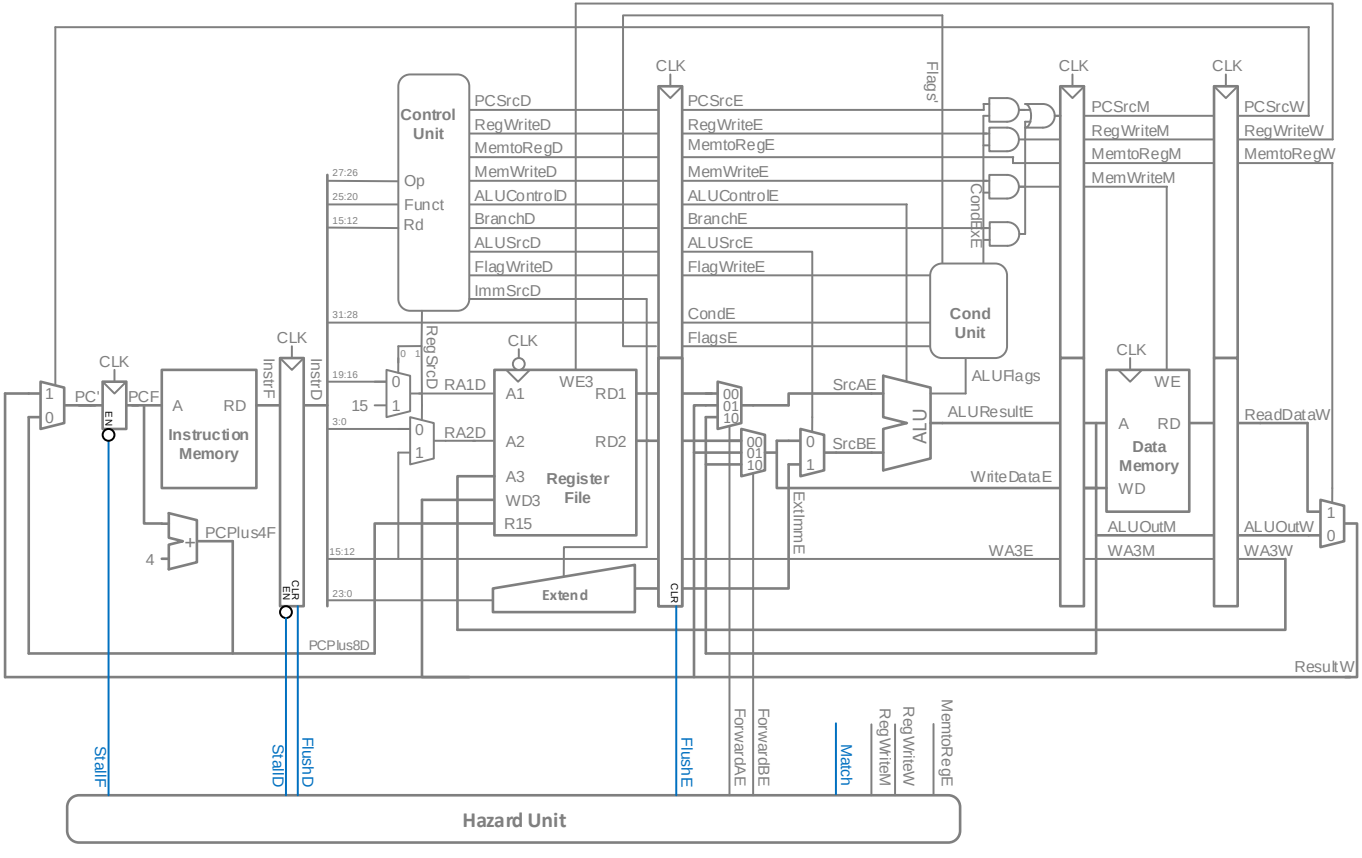
Stalling



Legger inn en stall «boble», som fryser buffrene et klokkeslag.
Merk at AND og ORR får 6 steg mot normalt 5 steg

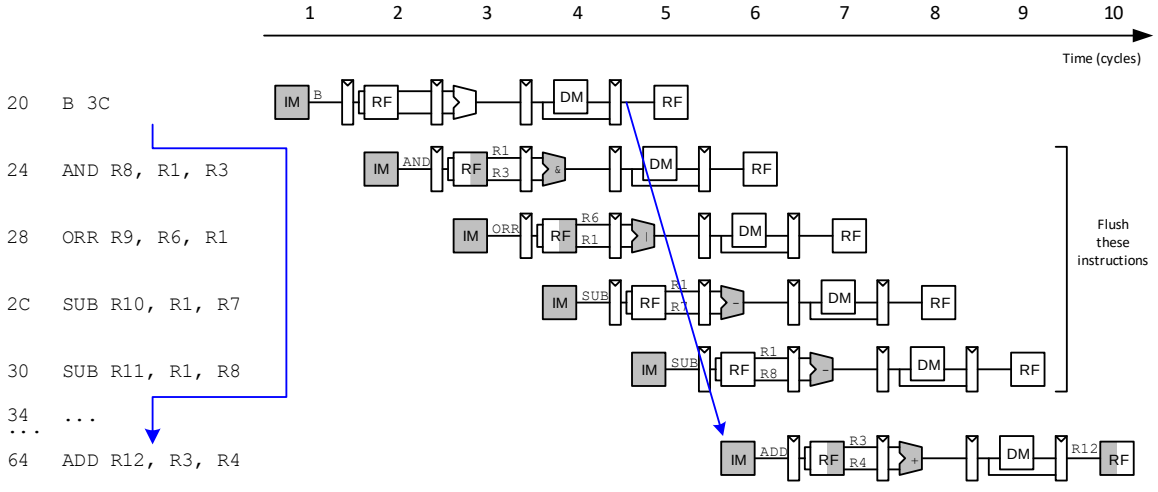
Stalling Hardware

Disable pipeline register
+
Flush/clear next register



Hvorfor hopper vi her til adresse 64?
BTA er gitt relativt til pc+8, dvs BTA = PC+8+offset = 20+8+3C = 64

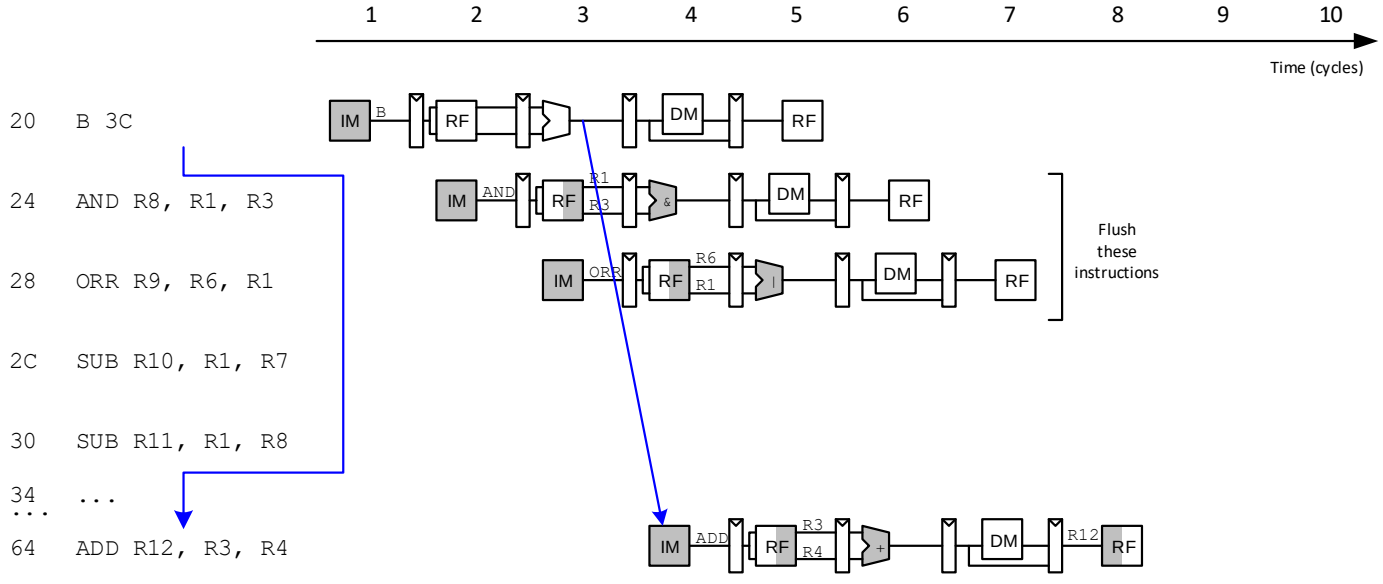
Control Hazards: Branch Misprediction Penalty



*Enkel predikasjon: at branch **aldri skal utføres***

- = Antall instruksjoner som vi må "flushe" når prediksjonen feiler. (4)
- Kan reduseres penalty ved å bestemme BTA tidligere
- Merk selv enkle «B 3C» må regne ut BTA = PC + 8 + offset.

Control Hazards with Early BTA



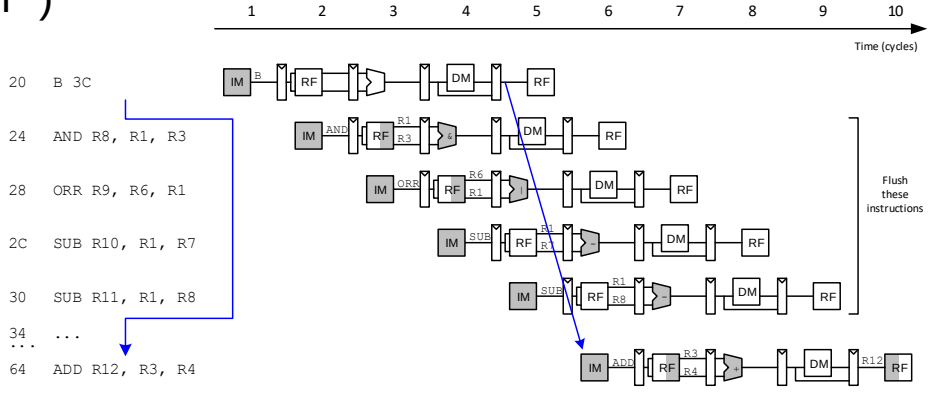
- Ved å *forwarde* BTA slipper vi å vente til instruksjonen er ferdig.
- Sparer 2 klokkesykler (slipper å flushe 2 instruksjoner)

Problemet med Branch

- Det tar tid å regne ut hvor vi skal branche siden vi alltid må regne ut BTA = PC+8+offset
- Det kan ta tid å finne ut om vi skal branche eller ikke.
  - F.eks. med BEQ må vi sjekke om Z verdien er høy (en verdi som kanskje ikke er klar enda).

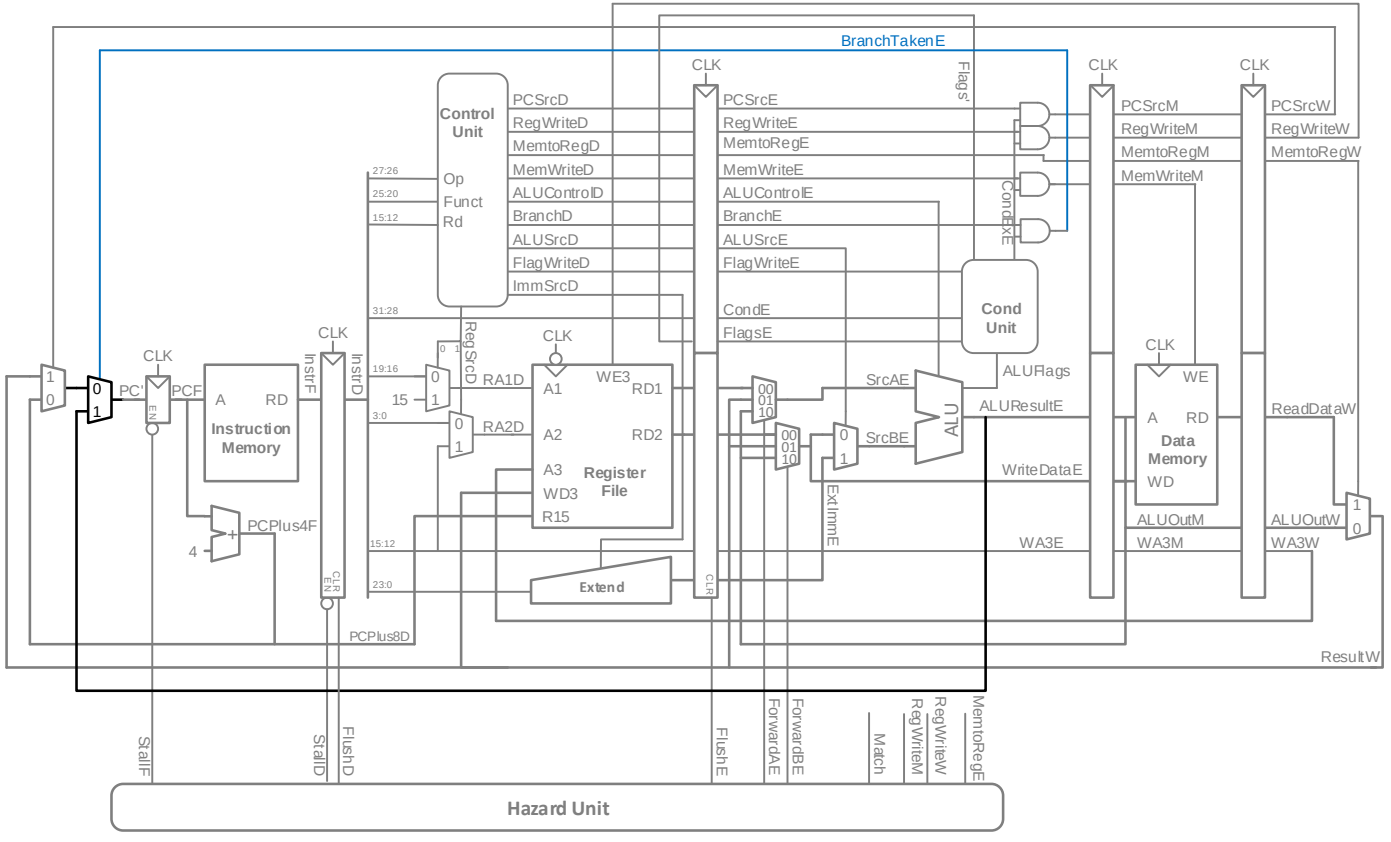
Branch – hvordan løse Control Hazards

- **Enkleste**. Vente til branch er avklart. Kostbart, vil alltid koste ekstra klokkesykler for hver eneste branch.
- **Bedre**. Predikere om vi skal branche, og begynne med å lese inn de predikerte instruksjonene. Dersom prediksjon var feil, må vi flushe disse.
  - Under har vi bare predikert at branch **aldri skal utføres** og lest inn de neste instruksjonene. (Ikke sett på betinget branch).
  - Vi kan benytte oss av mer avansert prediction. Også typisk å ha mer komplisert *fetch* steg slik at kan gå til den predikerte instruksjonen direkte. (7.7.3 Branch Prediction *)



Merk dette gjør vi for alle branch'es

Pipelined processor with Early BTA



Pipeline ytelse – hvordan finne CPI?

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type

  Kan forårsake *stalls*
- **Antar:**
  - 40% av loads blir brukt av neste instruksjon
  - 50% av branches blir feiltolket
- **Gjennomsnittlig CPI?**
  - Load CPI = 1 uten *stall*, 2 med *stall*
    - Gir **CPI_{lw}** = 1(0.6) + 2(0.4) = 1.4
  - Branch CPI = 1 uten *stall*, 3 med *stall*
    - Gir, **CPI_{beq}** = 1(0.5) + 3(0.5) = 2

Merk: gjelder for vår enkle 5-stegs pipeline implementasjon

CPI = (0.25)(1.4) + (0.1)(1) + (0.13)(2) + (0.52)(1) **= 1.23**

Pipeline ytelse eksempel

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60
Register file write	$t_{RFwrite}$	70

Critical path er *register read*: $T_{c3} = 2(t_{RFread} + t_{setup})$

$= 2[100 + 50]$ ps **= 300 ps** (3.33 GHz)

Pipeline ytelse eksempel

Program med 100 billion instruksjoner

- **Execution Time** = (# instruksjoner) $\times$ CPI $\times$ T_c
  = $(100 \times 10^9)(1.23)(300 \times 10^{-12})$
  = 36.9 sekunder

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	84	1
Multicycle	140	0.6
Pipelined	36.9	2.28

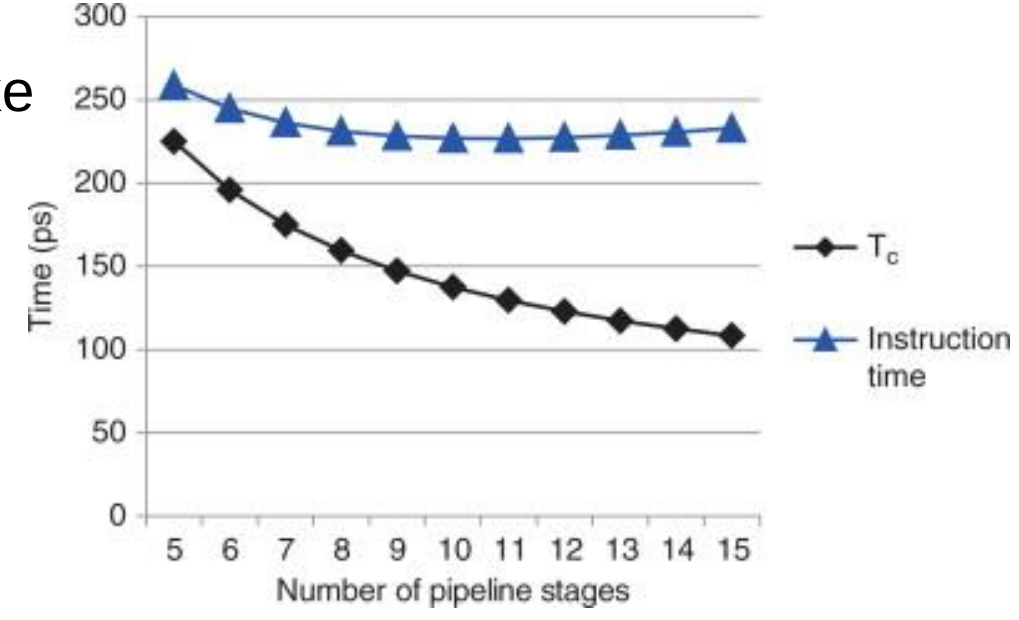
- Får 2.28x ytelse med pipeline.
  - Ikke 5 pga. *overhead, stalling og miss prediction penalty ++.*

Noen flere metoder fra Kap 7.7 (Orienteringstoff)

- Deep Pipelining
- Micro-operations
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

- “Letteste” måten å øke klokkehastigheten er å øke antall pipeline steg.
- Antall steg begrenset av
  - Pipeline hazards
  - Sequencing overhead
  - Strømforbruk
  - Kostnad
- 10-20 steg vanlig idag



Micro-operations

- Dele komplekse instruksjoner i mindre enkle instruksjoner
  - Såkalte *micro-operations* (*micro-ops* or *µ-ops*)
- Under kjøring blir komplekse instruksjoner tolket til *micro-ops*.
  - Forenkler den interne maskinvaren og muliggjør effektiv pipelining.
- Mye brukt på CISC (complex instruction set computer) arkitekturer.

- ARM eksempel:

Complex Op

```
LDR R1, [R2], #4
```

Micro-op Sequence

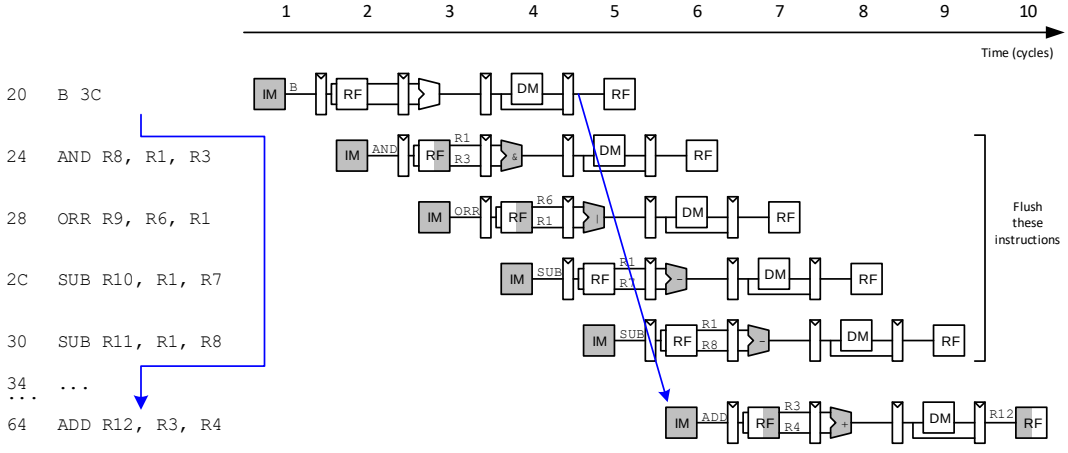
```
LDR R1, [R2]
ADD R2, R2, #4
```

Uten *µ-ops*, ville vi trengt 2 skrive innganger på RF.

Branch Prediction

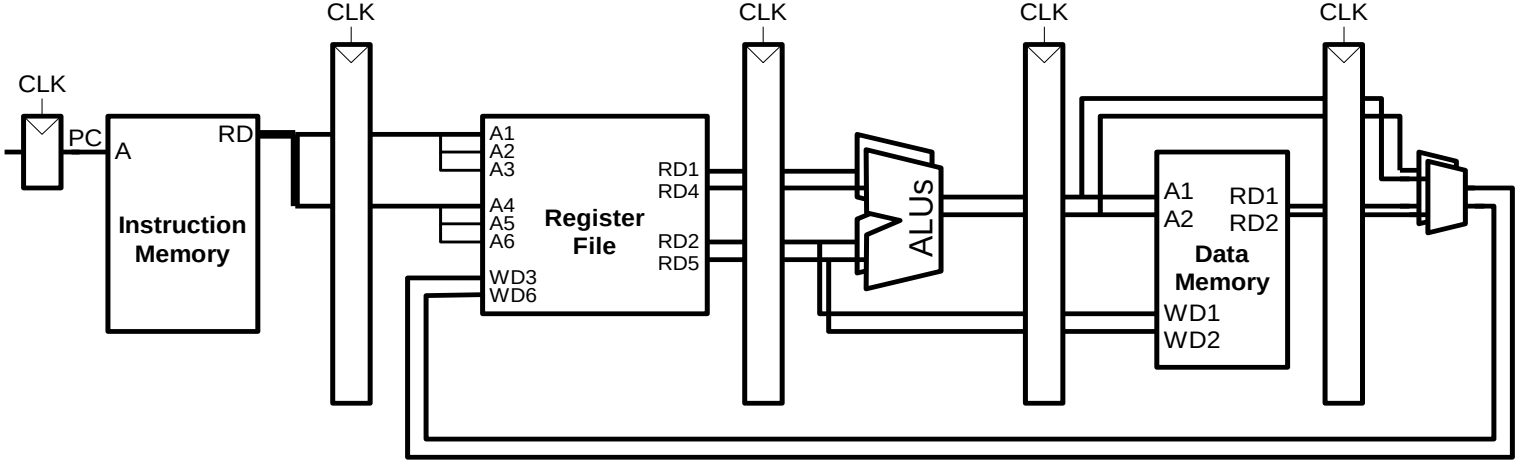
Kan prøve å gjette for å minimere flushing (misprediction)

- *Static branch prediction*: antar det som er vanlig (for/while løkker)
- *Dynamic branch prediction*: bruker historien for å gjette bedre.
  - God *branch prediction* kan oppnå 90% riktig predictions. (lavere CPI)



Superscalar

- Flere kopier av *datapath*, kan utføre *flere* instruksjoner samtidig.
- Men *avhengigheter* kan gjøre det vanskelig å starte flere instruksjoner på engang.
- Ideelt vil dobling av datapath gi *IPC* = 2 (CPI = 0.5)
  - Men grunnet kollisjoner/hazard, får vi i realiteten under 2.



Out-of-order Processors

- For å unngå problemet med *Hazards* og *stalls*, se etter *uavhengige* instruksjoner og starte dem så fort som mulig.
  - Sortere dem på avhengigheter.
- Vi kan forandre på rekkefølgen på instruksjonene, så lenge avhengigheter blir overholdt (koden blir korrekt utført).
- Ofte er problemet at vi ikke har nok registre å ta av for å unngå avhengigheter (R0-R14).
  - Kan utvide med *renaming registers* (f.eks. T0-T19) for å ha flere verdier i registre og unngå Hazards.
- Oppnår sjelden en *Instruction level Parallelism (*ILP) høyere enn 2-3 selv med 6x superscalar prosessor.

Single Instruction Multiple Data (SIMD)

- En enkelt instruksjon som kan utføre en operasjon på mange datapunkter på en gang
  - Vektorisering
- Typiske applikasjoner: GPU, Spillkonsoller, DSP, maskinlæring.
- Utfører korte aritmetiske operasjoner
  - *Packed arithmetic*
- Skreddersydd hardware for å effektiv kunne utføre spesifikke operasjoner.

Multithreading

Problem med Superscalar/Out-of-order processor, instruksjoner er sjelden parallelle og minne-fetching kan ta tid.

Multithreading: Dele programmet inn i deler som kan kjøre omhverandre (skifte til annen tråd når noe *staller*)

- *Context switching*: Skifte mellom *arkitektur tilstander*, og starte å kjøre et nytt sett med instruksjoner.
- *Multithreaded prosessor*: Utvider antall *arkitektur tilstander* (f.eks. 4x) slik at man kan skifte direkte mellom tråder uten forsinkelse. (Intel kaller dette hyperthreading)
    - Øker *throughput* siden vi utnytter ubenyttete ressurser ved stalling
    - «Billig» siden vi kun *replikerer* PC og RF

Multiprocessors

Bruke flere prosessor *kjerner* samtidig, med en type kommunikasjon mellom dem.

- **Symmetrisk / Homogene:** Like kjerner med delt minne
- **Heterogene:** Separate kjerner med forskjellige oppgaver, f.eks. CPU (big.LITTLE), GPU, kommunikasjon.
- **Clusters:** Hver kjerne har et eget sett med minne. Bruker *nettverk* som kommunikasjon (Datasentre, SETIATHOME)

Fungerer bra for parallelle problemer som er *enkle* å dele.

Men vanskelig å dele opp / programmere ikke parallelle problemer.

Oppsummering prosessor ytelse

- Stort marked – stort fokus på å maksimalisere ytelsen.
- Har visst noen avanserte teknikker som øker ytelsen
- Men økt ytelse betyr ofte økt pris og strømforbruk.
  - Cortex A15 – gir ~2.5x høyere hastighet enn Cortex A7
  - Men 6x mer strømforbruk
  - Mye dyrere ~10x (500 vs 50 kr)
- Merk at (størrelsen på transistoren*farbrikasjons-prosessene*) er viktig for ytelse og strømforbruk.
  - I 2026 masseproduseres prosessorer med 2 nm-teknologi, og industrien arbeider mot 1,4 nm og videre mot 1 nm.
  - Men nm-tallene er i dag teknologinavn – ikke transistorenes faktiske størrelse.

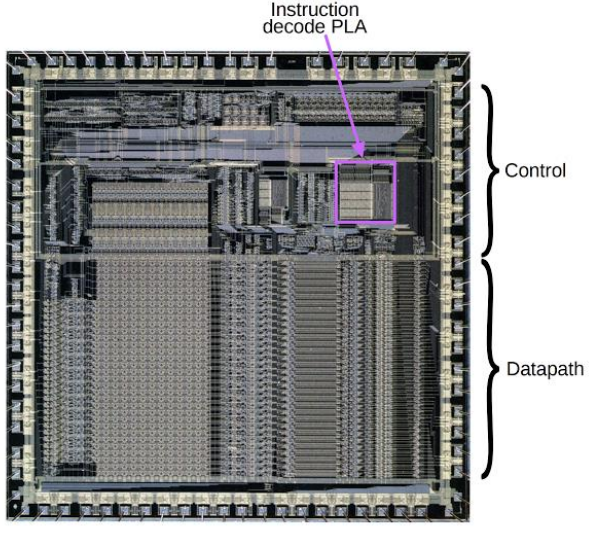
MOSFET scaling (process nodes)

20 µm – 1968
10 µm – 1971
6 µm – 1974
3 µm – 1977
1.5 µm – 1981
1 µm – 1984
800 nm – 1987
600 nm – 1990
350 nm – 1993
250 nm – 1996
180 nm – 1999
130 nm – 2001
90 nm – 2003
65 nm – 2005
45 nm – 2007
32 nm – 2009
22 nm – 2012
14 nm – 2014
10 nm – 2016
7 nm – 2018
5 nm – 2020
3 nm – 2022
2 nm – 2025

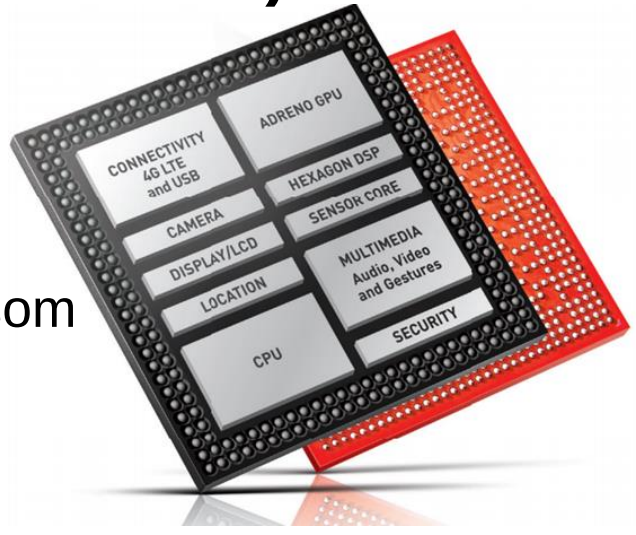
Future
1 nm ~ 2027

ARM - Stiftet av Acorn og Apple i 1990

- ARM1 kom ut i 1985 (Acorn).
- Fokus på strømgjerrig design. Stor suksess innenfor *embedded systems* (1994).
- ARM startet å lisensiere sine kjerner som *hard macro (utlegg)* eller *soft macro (verilog)*.
  - Et valg som ga stor suksess.
  - Ga tid og midler til å videreutvikle arkitekturen sammen med kunder.
- Nokia 6110 (1997), første mobiltelefon med ARM prosessor – massiv suksess.
- ARM teknologi blir i dag brukt i mer enn 350 milliarder brikker.



System on Chip (Heterogen multiprosessor)



- SoC: «Komplett datamaskin» på en chip, kombinerer flere moduler.
  - CPU'er, GPU, WIFI, modem (4G/5G), mm.
- Med integrasjonen øker vi ytelsen samtidig som man minker strømforbruket og størrelse.
  - Men dyrere og mer komplisert design.
- ARM svært mye brukt i SoC design.
  - «Alle» mobiltelefoner i dag er bygd rundt en ARM prosessor (Apple og Android)
  - Spillkonsoller og biler.
  - Raspberry PI
    - CPU (4× ARM Cortex-A53, 1.4GHz), GPU, Ethernet, USB, I/O, osv.

Historiske utviklingen av ARM

Dhrystone MIPS / Mhz ~ IPC

c	Year	Architecture	Pipeline Depth	DMIPS/MHz	Representative Frequency (MHz)	L1 Cache	Relative Size
ARM1	1985	v1	3	0.33	8	N/A	0.1
ARM6	1992	v3	3	0.65	30	4 KB unified	0.6
ARM7	1994	v4T	3	0.9	100	0–8 KB unified	1
ARM9E	1999	v5TE	5	1.1	300	0–16 KB I+D	3
ARM11	2002	v6	8	1.25	700	4–64 KB I+D	30
Cortex-A9	2009	v7	8	2.5	1000	16–64 KB I+D	100
Cortex-A7	2011	v7	8	1.9	1500	8–64 KB I+D	40
Cortex-A15	2011	v7	15	3.5	2000	32 KB I+D	240
Cortex-M0+	2012	v7M	2	0.93	60–250	None	0.3
Cortex-A53	2012	v8	8	2.3	1500	8–64 KB I+D	50
Cortex-A57	2012	v8	15	4.1	2000	48 KB I+32 KB D	300
Cortex-X1	2020	v8.2-A	~13	~10?	2800-3000	64 KB I+64 KB D	450

En forberedelse på 12x i IPC og 250x i klokkefrekvens på 30 år.
Cortex-M0+ er “ekstremt strømgjerrig”, veldig enkel 2 pipeline. Under 10 µW/MHz.

Oppsummering

- Kort oppsummering av Single og multi-cycle.
- Pipelining
  - *Temporal parallellisme* gir bedre ytelse
  - Sett på forenklet pipeline versjon med 5 steg.
  - Problem: Data og control *Hazards*
  - Løsninger: Stall, forwarding, Early BTA
- * Noen avanserte metoder
- * Litt om diverse om ARM

- **Ukeoppgaver**: 7.28, 7.30, 7.32 og 7.37