

UiO : Department of Informatics
University of Oslo

Forelesning 7a IN2060

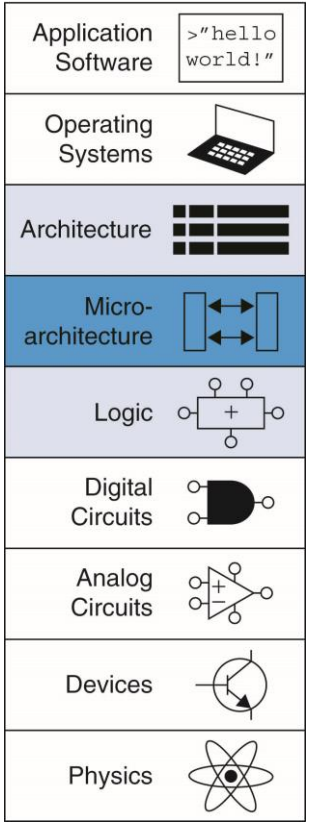
Mikroarkitektur del a



Mikroarkitektur

- Hva menes med mikroarkitektur?
- Litt om ytelse og hvorfor vi har forskjellige mikroarkitekturer.

1. Single-Cycle Processor
2. Multicycle Processor
3. Pipelined Processor (neste uke)



Introduksjon

Vi har lært det vi trenger for å implementere en ARM arkitektur.

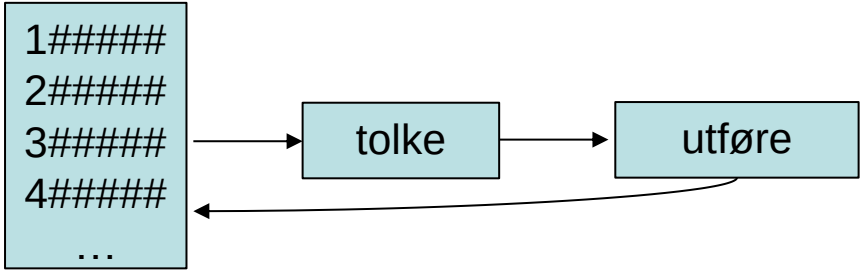
- Kombinatorisk og synkron logikk.
- Verktøy: Tilstandsmaskiner og digitale byggeblokker
- Spesifisert arkitekturen (ARM), som er definert av
  - Instruksjonene som må støttes
  - Arkitektur tilstandene (R0-R15 og *status*-registeret)

Hvordan implementeres dette i Hardware?

- Arkitekturen spesifiserer ikke hvordan prosessoren skal implementeres. Kun hva som skal **støttes**.

Definisjon

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



- Basert på nåværende *arkitektur tilstand*, utfører CPU en spesifikk instruksjon med spesifikke data, og produserer en ny *arkitektur* tilstand.

Ytelse

- Løsningen - et kompromiss mellom
  - Kostnad
  - Strømforbruk
  - Hastighet (hvor *raskt* den faktisk jobber)
- Klokkefrekvensen bestemmer ikke ytelsen alene
  - En detaljert analyse er nødvendig.
- *Benchmarks* = Ytelsestester.
  - Programmer som kan brukes for å måle ytelsen.
  - Må velge den som er mest relevant.

Eksempel på mikroarkitekturer

Cortex-A53 og *Cortex-A72* er begge implementasjoner av *Armv8-A* arkitekturen. Dvs. basert på samme arkitektur, men forskjellig implementert.

	Cortex-A53	**Cortex-A72**
Target	Optimized for **power efficiency**	Optimized for **performance**
Pipeline	8 stages In-order	15+ stages Out-of-order
Caches	L1 I cache: 8KB - 64KB L1 D cache: 8KB - 64KB L2 cache: optional, up to 2MB	L1 I cache: 48KB fixed L1 D cache: 48KB fixed L2 cache: mandatory, up to 2MB

Maskinkode/programkode som er *arkitektur-kompatibel* med *Armv8-A*, kan kjøre på Cortex-A53 og Cortex-A72 uten modifikasjoner, siden begge implementerer samme arkitektur.

Mikroarkitektur

- Mål: Hvordan lage en implementasjon som utnytter chip arealet best mulig.

- Skal se på 3 *forenklete* versjoner (mer info kap. 7.7 og 7.8).
- Detaljene skal gi dere et innblikk i:
  - Hvordan en CPU utfører instruksjoner/programmer.
  - De valgene vi må ta når vi skal implementere en CPU.
  - Hvordan tolke ytelsen til en CPU.

Mikroarkitektur- *Datapath* og *Control unit*

- Løsningen kan deles opp i to deler
  - *Datapath*: Funksjonelle deler (ALU, Minne/registre, osv)
    - Opererer på WORD av data (32 bit) - *dataflyten*
  - *Control unit*: Funksjonene som kontrollerer flyten
    - *Control unit* tolker instruksjonene og forteller datapath'en hvordan den skal utføre dem.
    - Basert på kombinatorisk logikk / tilstandsmaskiner
    - Kontrollerer MUX'er, Minne, ALU osv.

Mikroarkitektur – Forskjellige implementasjoner

Vi skal implementere en arkitektur på 3 forskjellige metoder

- **Single-cycle:**
  - Instruksjonene blir utført i løpet av *en* klokkesykel.
- **Multicycle:**
  - Instruksjonene blir delt inn i flere kortere deler/steg.
- **Pipelined:**
  - Instruksjonene blir delt inn i flere kortere deler/steg.
  - Flere steg kan kjøre samtidig (parallelt).
    - Sette i gang nye instruksjoner før den forrige er ferdig.

«Alle» kommersielle CPU bruker idag pipelining.

Forenklet ARM arkitekturen

Vi skal kun ta for oss en lite utvalg av ARM instruksjoner:

- **Dataprosessering:**
  - `ADD, SUB, AND, ORR`
  - med register og immediate Src2, ingen *shifts*.
- **Minnehåndtering:**
  - `LDR, STR`
  - med *positive immediate* offset
- **Branch:**
  - `B`

Tilstrekkellig for å få forståelse.
Ukeoppgaver: Utvide med flere instruksjoner.

Arkitekturens *tilstandselementer*

Inneholder *tilstanden* til prosessoren

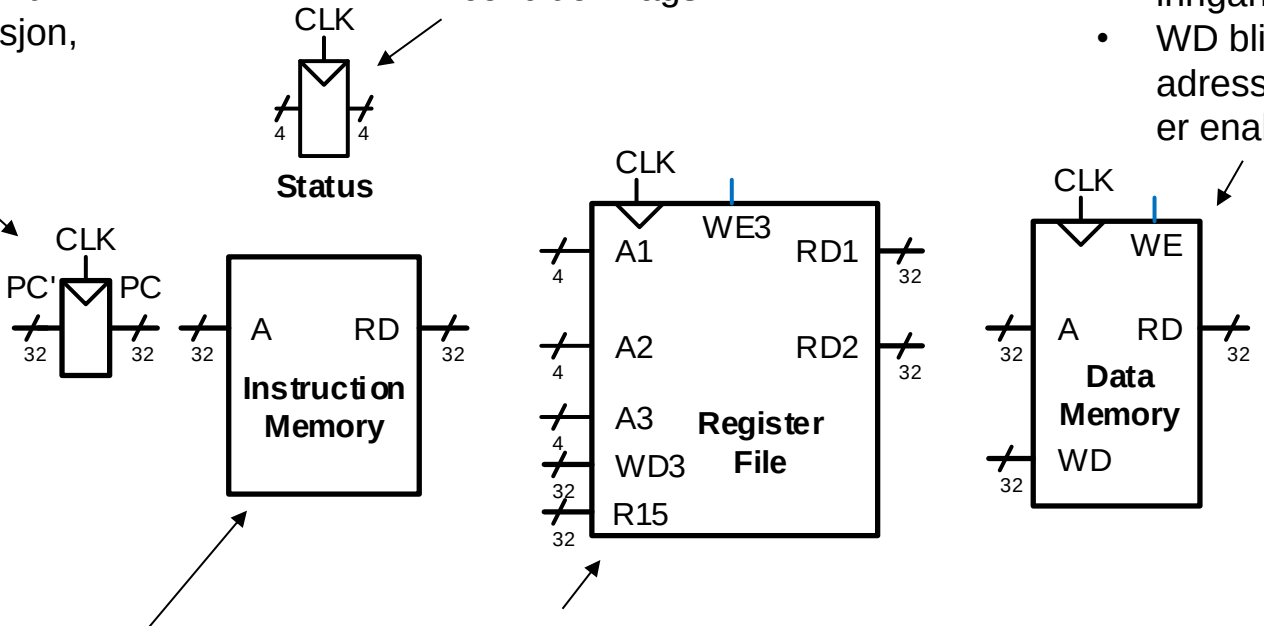
- Arkitektur-tilstand:
  - 16 registere (inklusiv PC)
  - Status registeret
- Minne

Arkitekturens tilstandselementer

PC gir adressen til aktuelle instruksjon, og PC' til neste instruksjon.

Status registeret inneholder ALU condition flags

- Kan hente ut data med inngang A.
- WD blir skrevet til adresse A dersom WE er enablet.

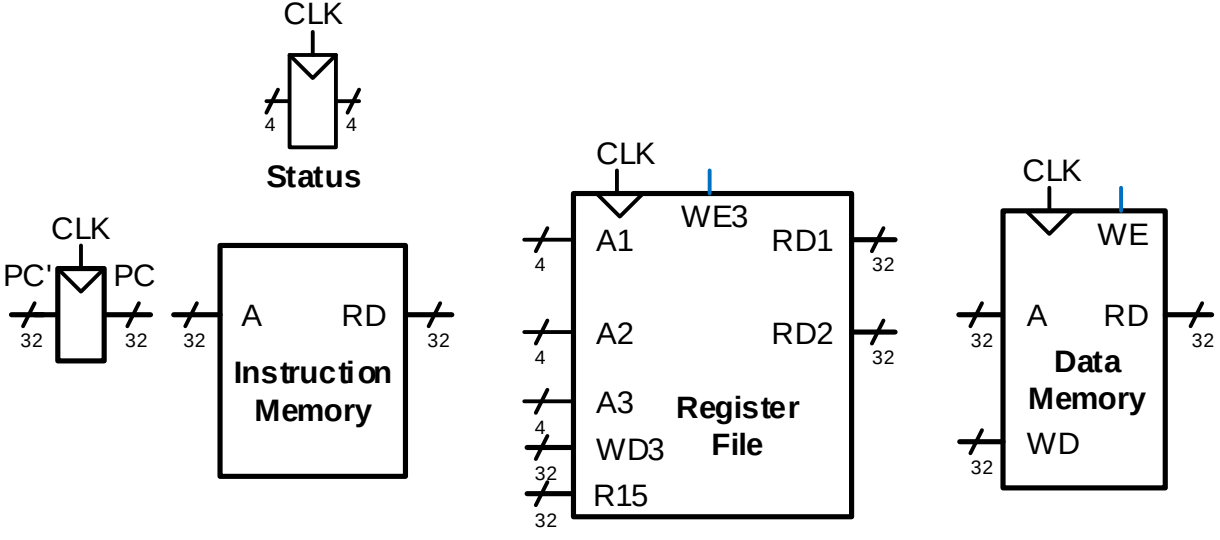


IM henter riktig instruksjon basert på 32-bit PC adressen.

- RF inneholder R0-R15. Leses ut via adresser A1 og A2.
- Hvis WE3 høy, skrives 32-bit verdien WD3 til register adresse A3. (R15/PC oppdateres kontinuerlig.)

Single-Cycle ARM Processor

- Dette er elementene vi trenger for å sette sammen en enkel prosessor.
- Merk alle unntatt IM er styrt av klokkeflanke
  - *Single-cycle*: instruksjonen skal bli ferdig i løpet av en klokkeperiode.
- Vi begynner med dataflyten (*datapath*).
  - Bygger opp kretsen ved å implementere instruksjon for instruksjon.

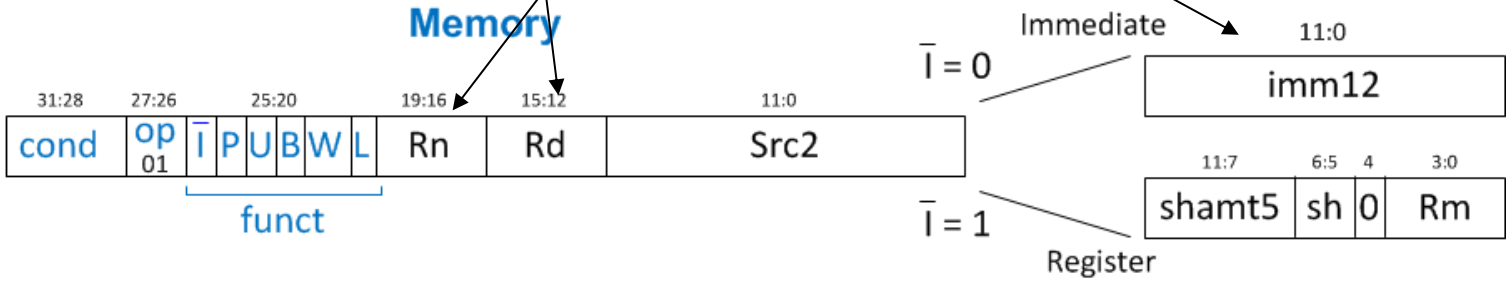


Single-Cycle ARM Processor

- **Datapath:** begynner med `LDR` instruksjonen

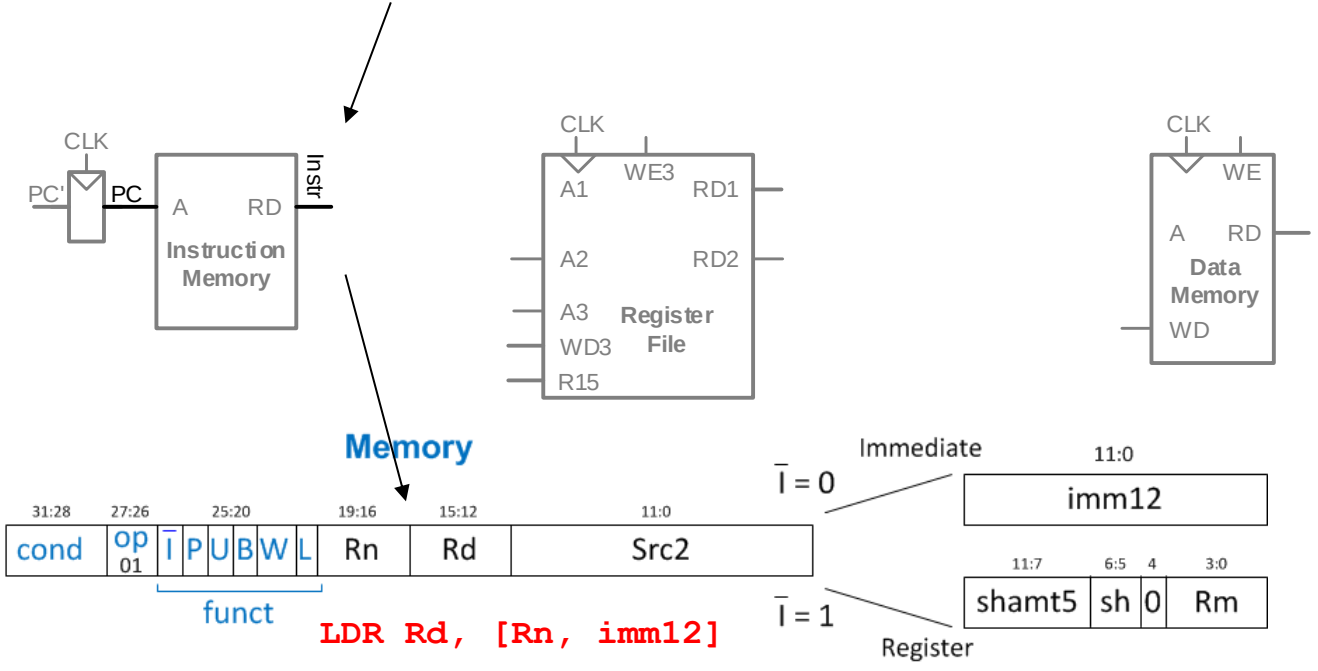
- **Eksempel:** `LDR R1, [R2, #5]`
  `LDR` **`Rd, [Rn, imm12]`**

Merk. Vi skal hente ut verdien *mem(Rn+imm12)* og skrive verdien til register adresse *Rd*.



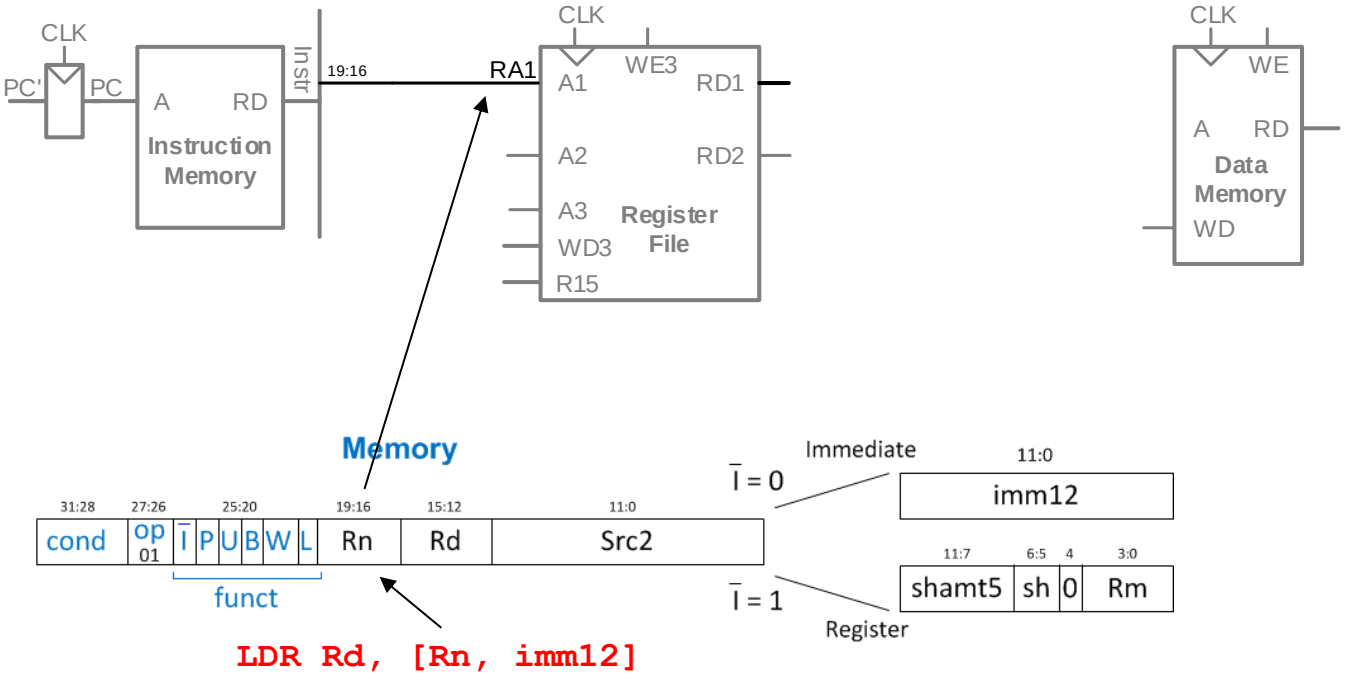
Single-Cycle Datapath: *LDR fetch*

- **Steg 1:** Hent instruksjon fra IM (*fetch*)
  - Etter kort tid vil 32-bit *Instr* bussen reflektere `LDR` instruksjonen.



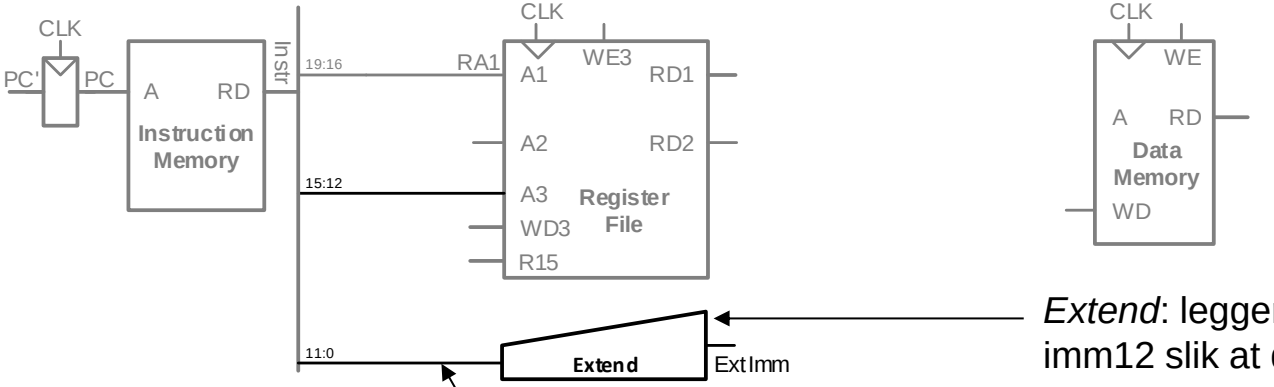
Single-Cycle Datapath: *LDR Reg Read*

- **Steg 2:** Hent ut baseadressen fra registeret .

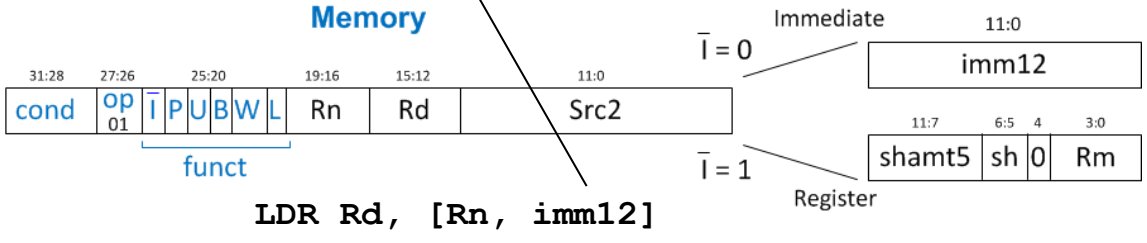


Single-Cycle Datapath: *LDR Immediate*

- **Steg 3:** Utvid 12-bit immediate verdi til 32-bit (extend)



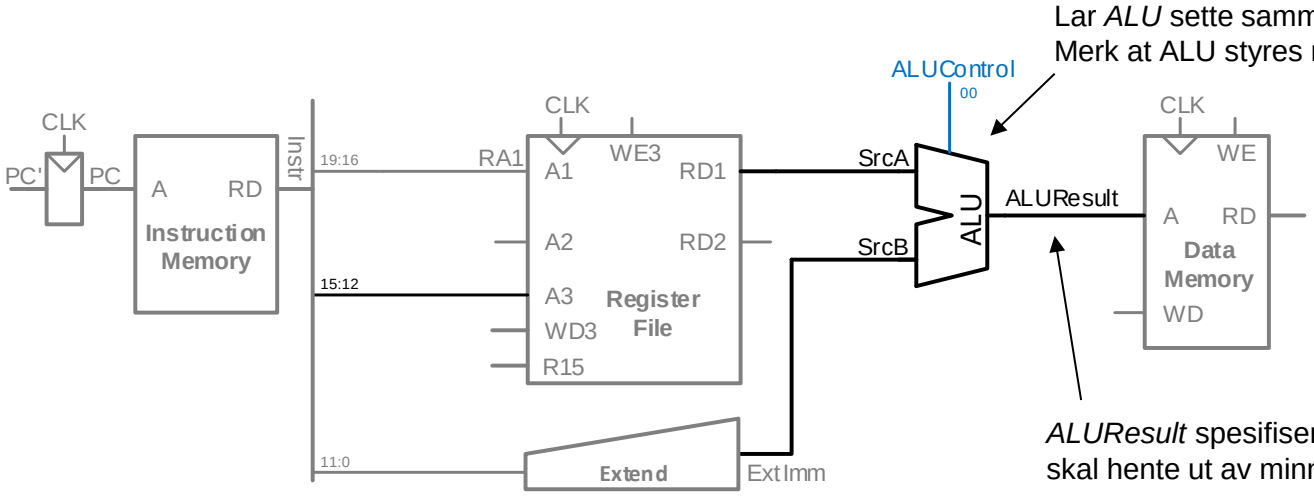
Extend: legger til nuller foran imm12 slik at det blir på 32-bit

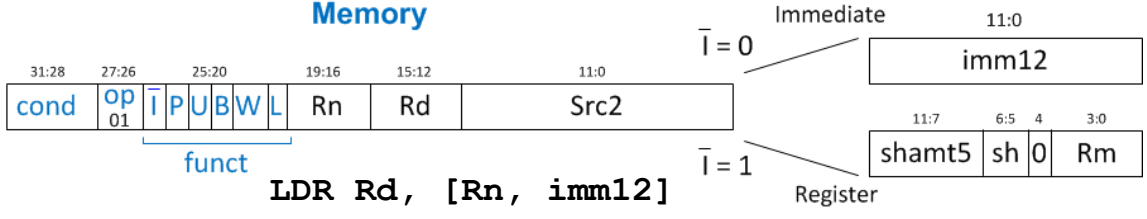



```
LDR Rd, [Rn, imm12]
```

Single-Cycle Datapath: LDR Address

- **Steg 4:** Beregn minneadresse (baseadresse + offset)



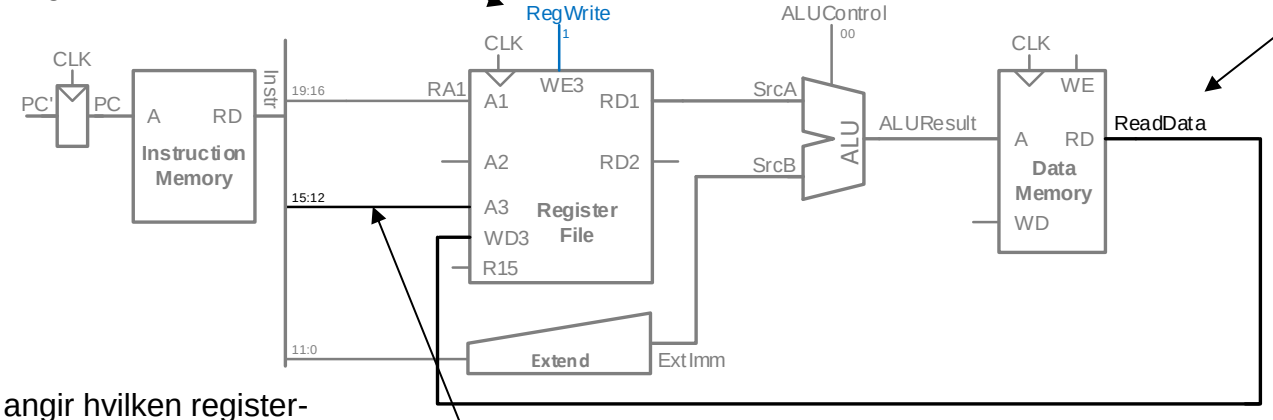


Single-Cycle Datapath: *LDR Mem Read*

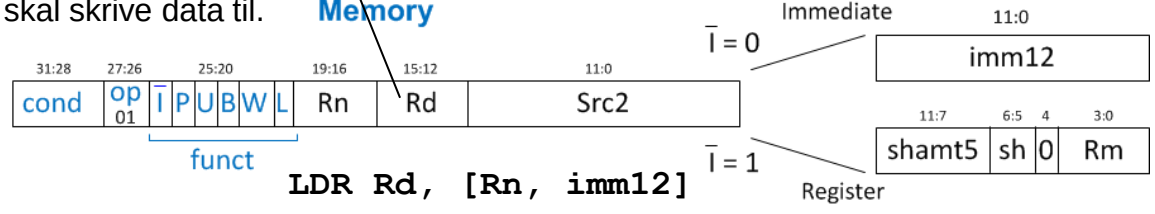
- **Steg 5:** Hent data fra minne og skriv det tilbake til registeret.

Vi må sørge for at instruksjonen setter RegWrite til 1.

Det vi har lest ut av minne sendes tilbake til registeret



A3 angir hvilken register-variabel vi skal skrive data til.

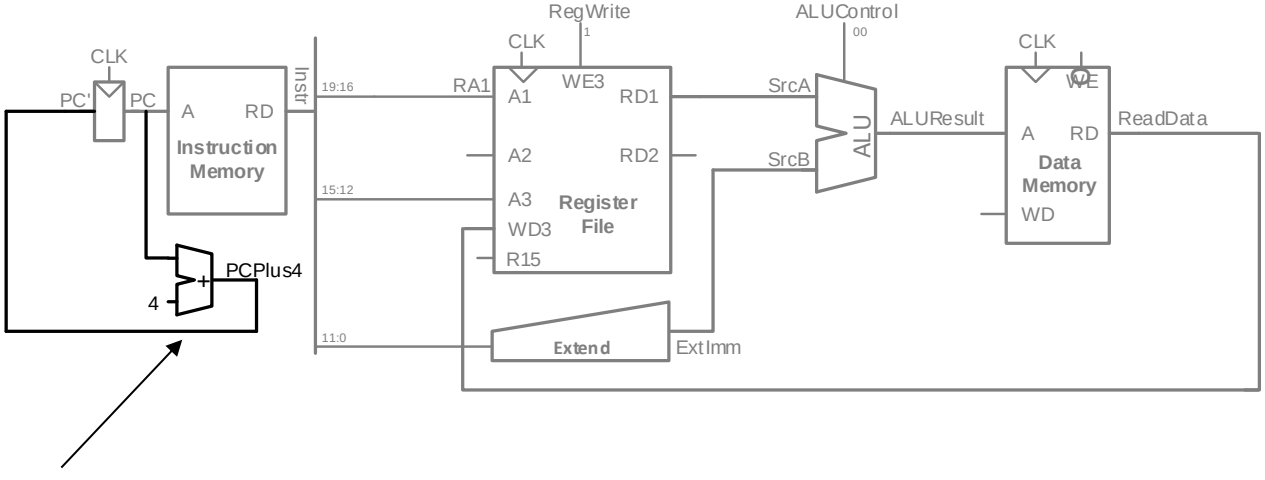


LDR Rd, [Rn, imm12]

- Register Rd vil nå få ny verdi ved neste klokkeflanke.
- Vi har utført instruksjonen. Hva gjenstår?
- Vi må oppdatere PC.

Single-Cycle Datapath: Øke PC med 4

- **Steg 6:** Beregn addresse til neste instruksjon

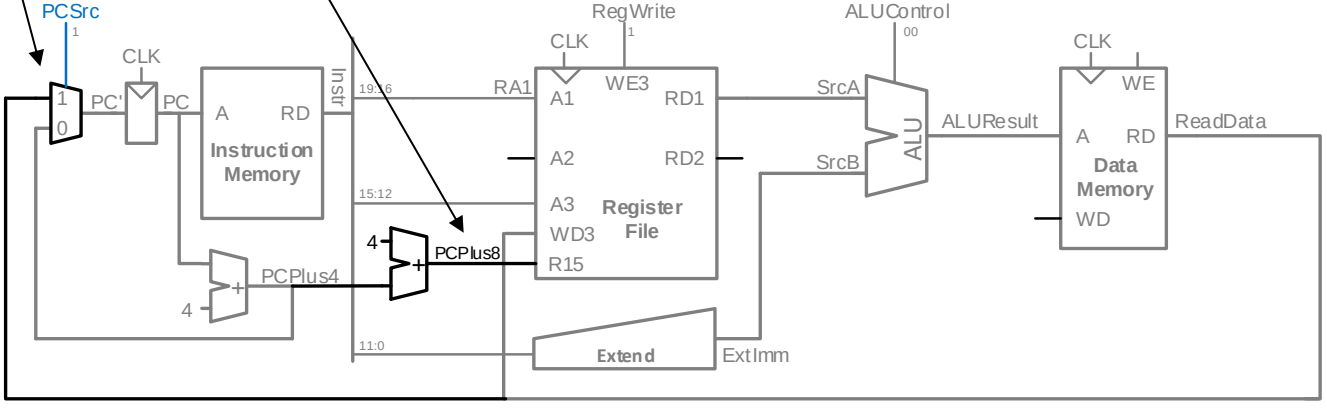


- Legger til en adder som legger til 4 (= word).
- PC vil bli oppdatert til neste instruksjon ved positiv klokkeflanke.
- Men hva med *branch*?

Single-Cycle Datapath: Instruksjoner trenger tilgang til PC

For å kunne *branche* til vilkårlig adresse, må instruksjoner kunne sette PC og vite verdien til PC.

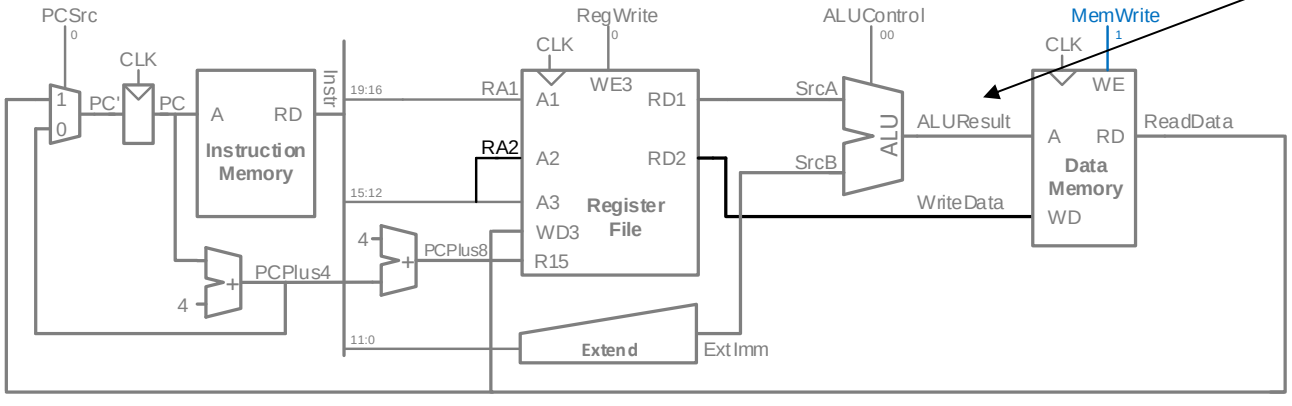
- R15 blir *kontinuerlig* oppdatert med verdien PC+8 (historisk).
- Vha. en Mux kan vi sette verdien til PC direkte.



Single-Cycle Datapath: `STR`

La oss utvide **datapath** til å støtte `STR`:

- Lagre register verdien *Rd* til adressen *Rn+imm12*.

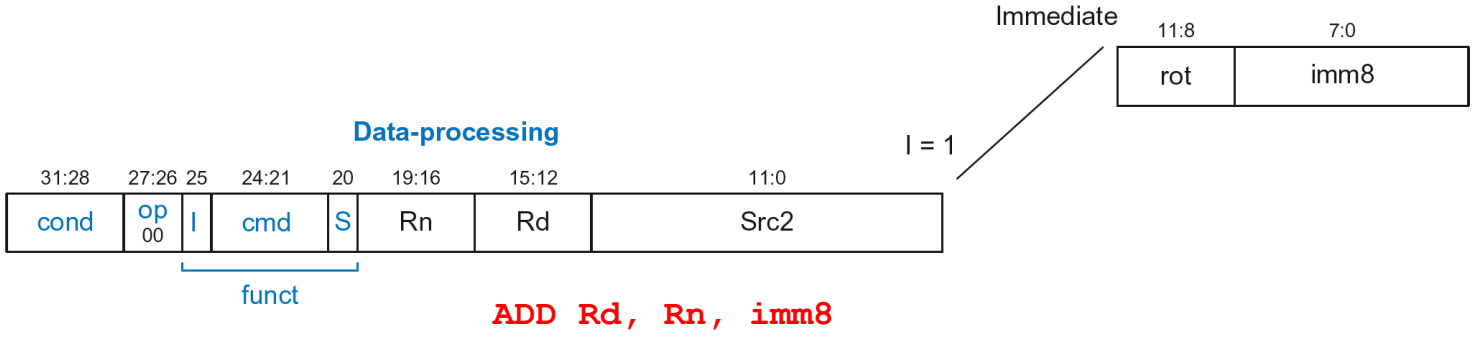




- Regner oss fram til riktig adresse som sist med `LDR`.
- Rd spesifiserer hvilken registerverdi vi skal lagre til minnet.
- STR må sørge for å sette MemWrite = 1.
- Men også Regwrite=0 for å ikke tukle med registeret.

Single-Cycle Datapath: Data-processing

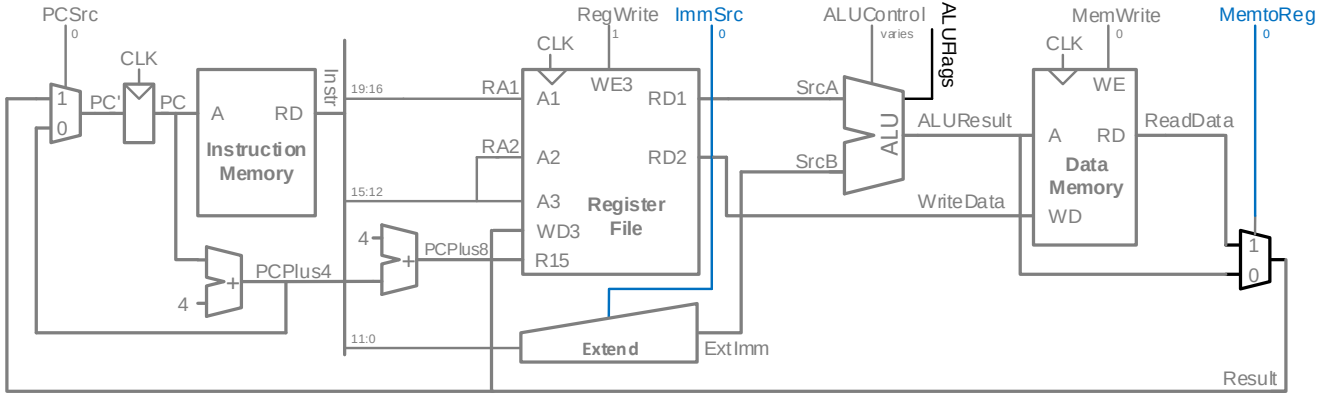
- Skal utvide datapath'en til å støtte dataprosesserings-instruksjoner med immediate Src2
  - Dvs. ADD, SUB, AND, ORR.
- Eksempel: **`ADD Rd, Rn, imm8`**



Merk: Vi kan gjenbruke mye logikk fra LDR siden instruksjonsformatet er såpass likt.

Single-Cycle Datapath: Data-processing

- Hent ut `Rn` verdi og *zero-extend* `Imm8`
  - *ImmSrc* velger om vi skal zero-extend `Imm8` eller `Imm12`
- Gjenbruker ALU'en som vi brukte til å regne ut adresse.
- Skriver *ALUResult* til registeret

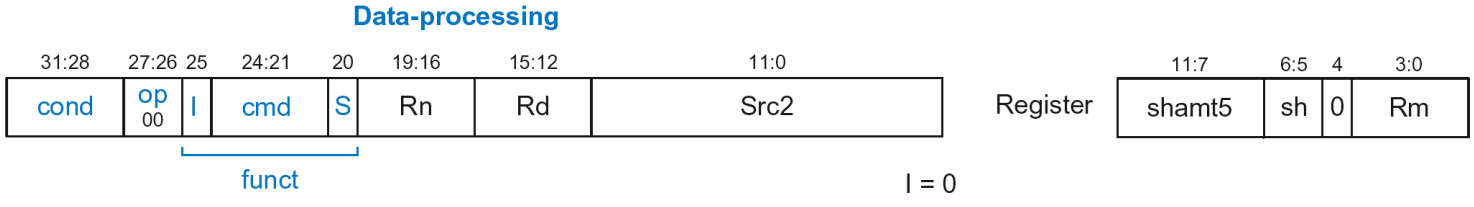


`ADD Rd, Rn, imm8`

Single-Cycle Datapath: Data-processing

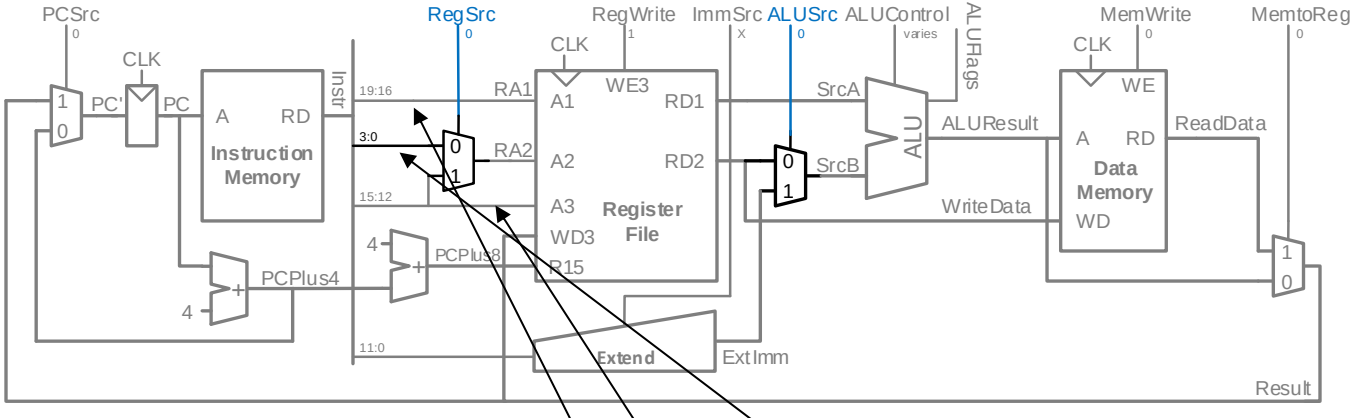
Med register som Src2

- **Eksempel: `ADD Rd, Rn, Rm`**
  - **Skal addere sammen to register variabler og lagre summen til registeret.**

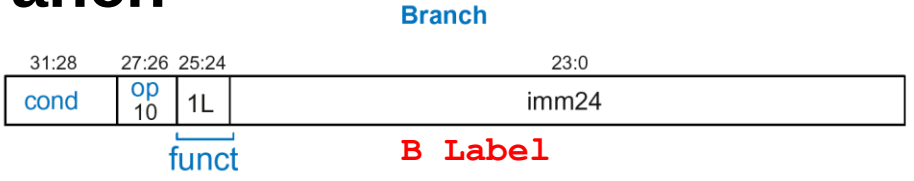


Single-Cycle Datapath: Data-processing

- Les ut verdiene `Rn` and `Rm`, og legg sammen.
- Skriv *ALUResult* til registeret.
  - Til register variable `Rd`

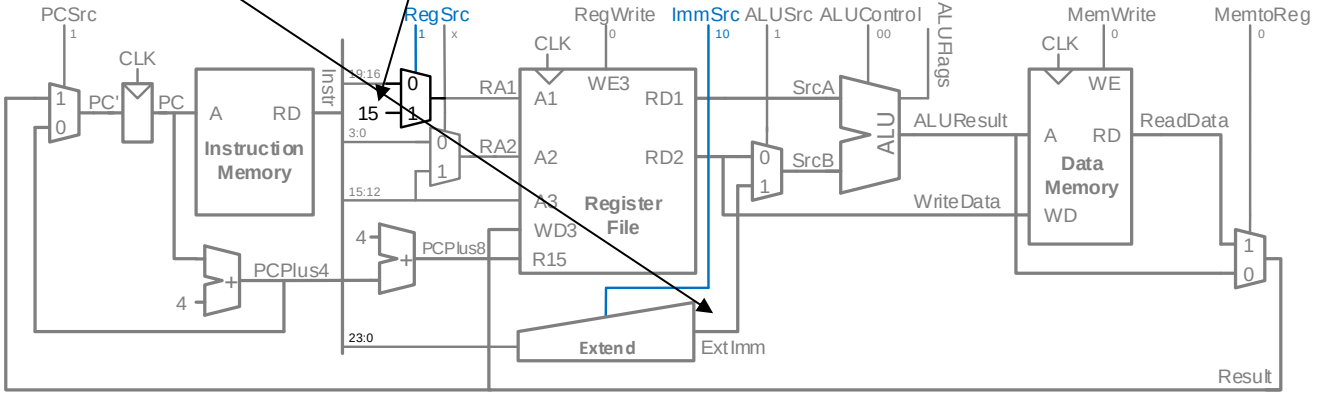


Single-Cycle Datapath: Branch



Skal implementere **B.**

- Må beregne BTA (*branch target address*) og skrive til PC
  - BTA = offsett + (PC + 8)
  - offsett = *(*Imm24 *<< 2)* og *sign-extended*

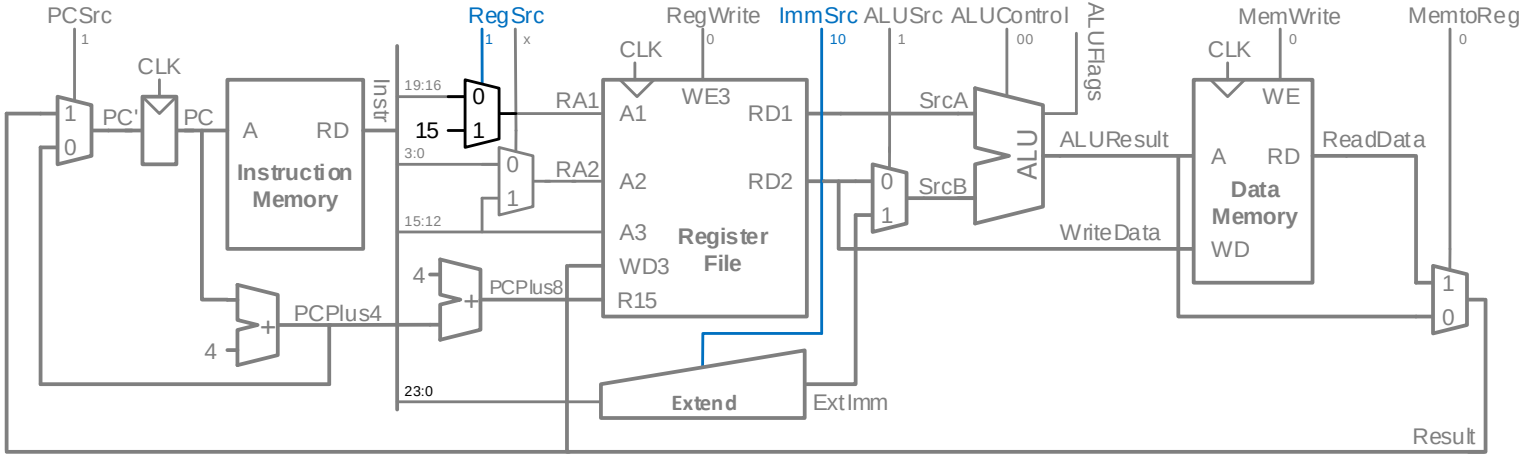


Men hvordan holde orden på alle kontrollsignalene?

Single-Cycle Datapath: ExtImm

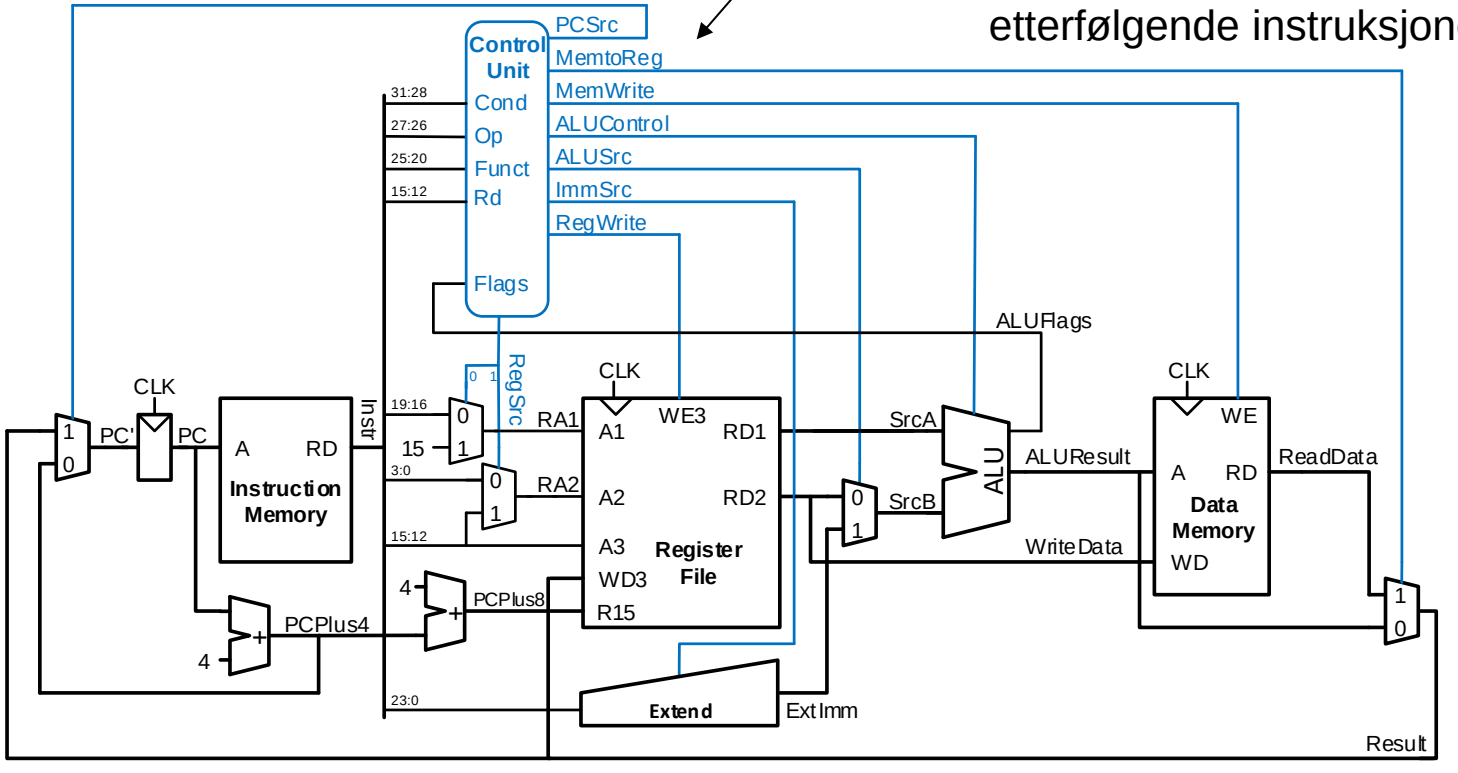
- Extend funksjonaliteten er typisk avansert for å kunne maksimalt utnytte *immediate bit'ene.*

$ImmSrc_{1:0}$	ExtImm	Description
00	{24{0}, $Instr_{7:0}$}	Zero-extended *imm8*
01	{20{0}, $Instr_{11:0}$}	Zero-extended *imm12*
10	{6{$Instr_{23}$}, $Instr_{23:0}$} 00	Sign-extended *imm24*

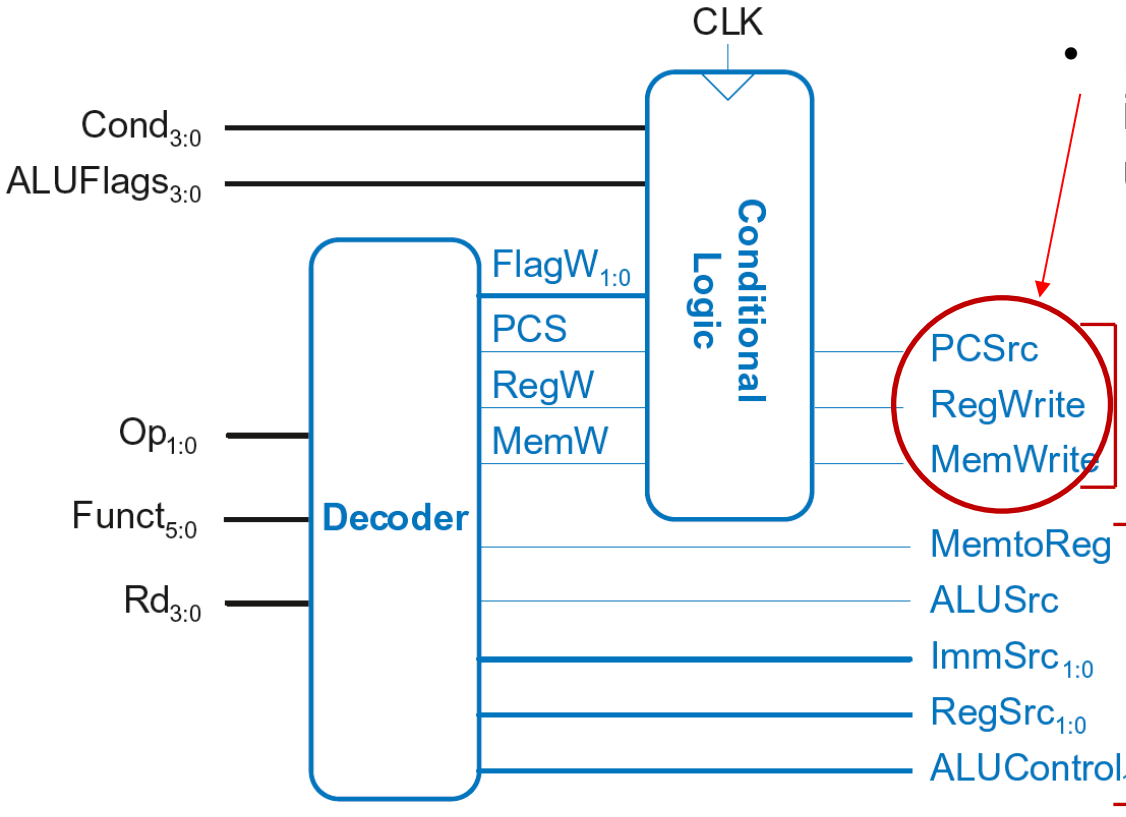


Single-Cycle: *Control Unit*

- Kontrollsignalene vi trenger
- Mesteparten er direkte tolket fra instruksjonen
- Utregnet vha. kombinatorisk logikk.
- Lagrer også unna *ALUFlags* til etterfølgende instruksjoner.



Single-Cycle Control



- Disse forandrer på **tilstanden** (PC, RF, Memory)
- Må settes til 0 dersom instruksjonen *ikke* skal utføres (betingelse).

Må først innom Conditional Logic, før vi sender til datapath

Sendes direkte til datapath

Eksempel: ORR

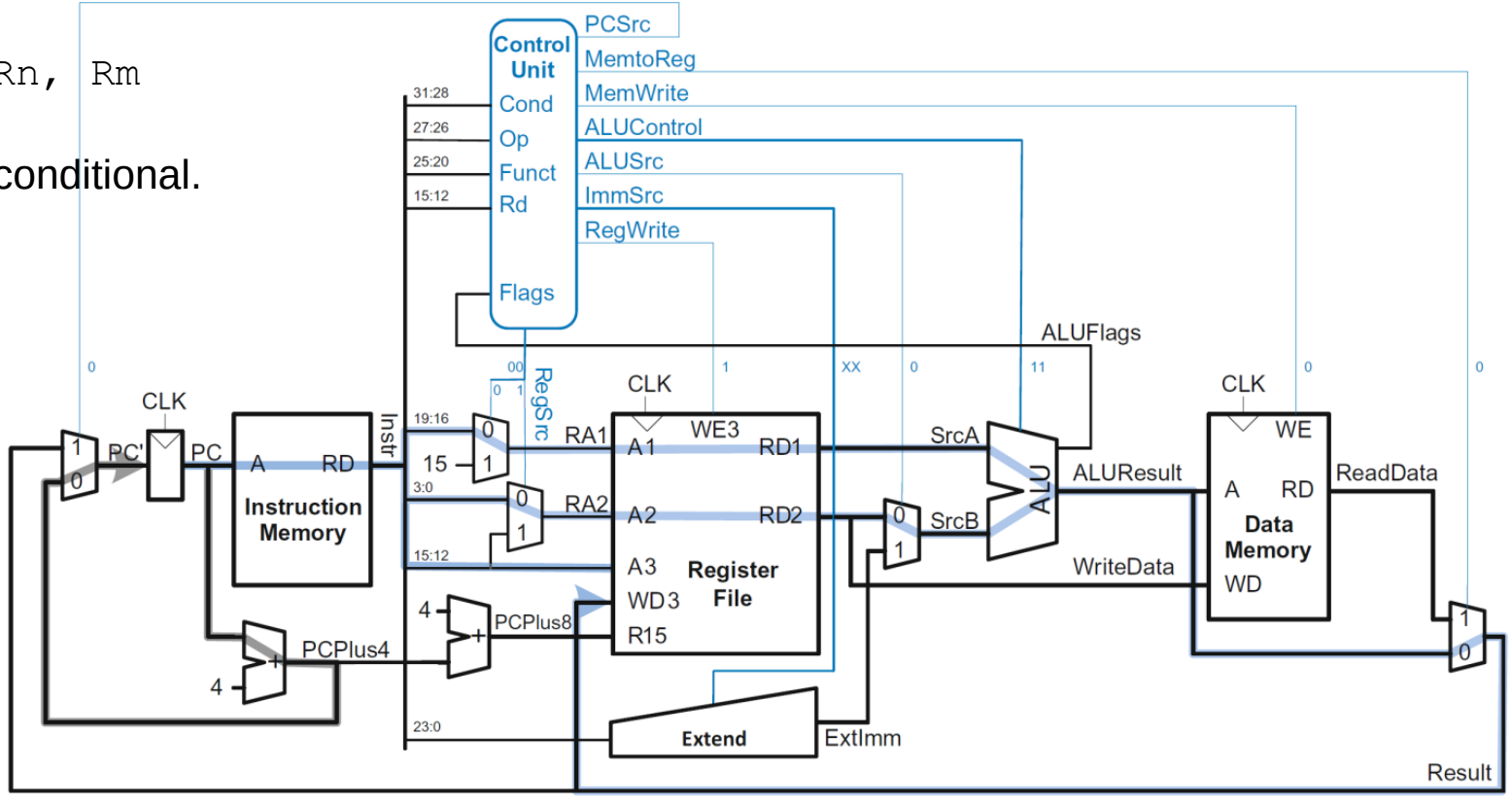
Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1

Eksempel på hvilke kontrollsignaler vi trenger.

Bitwise OR

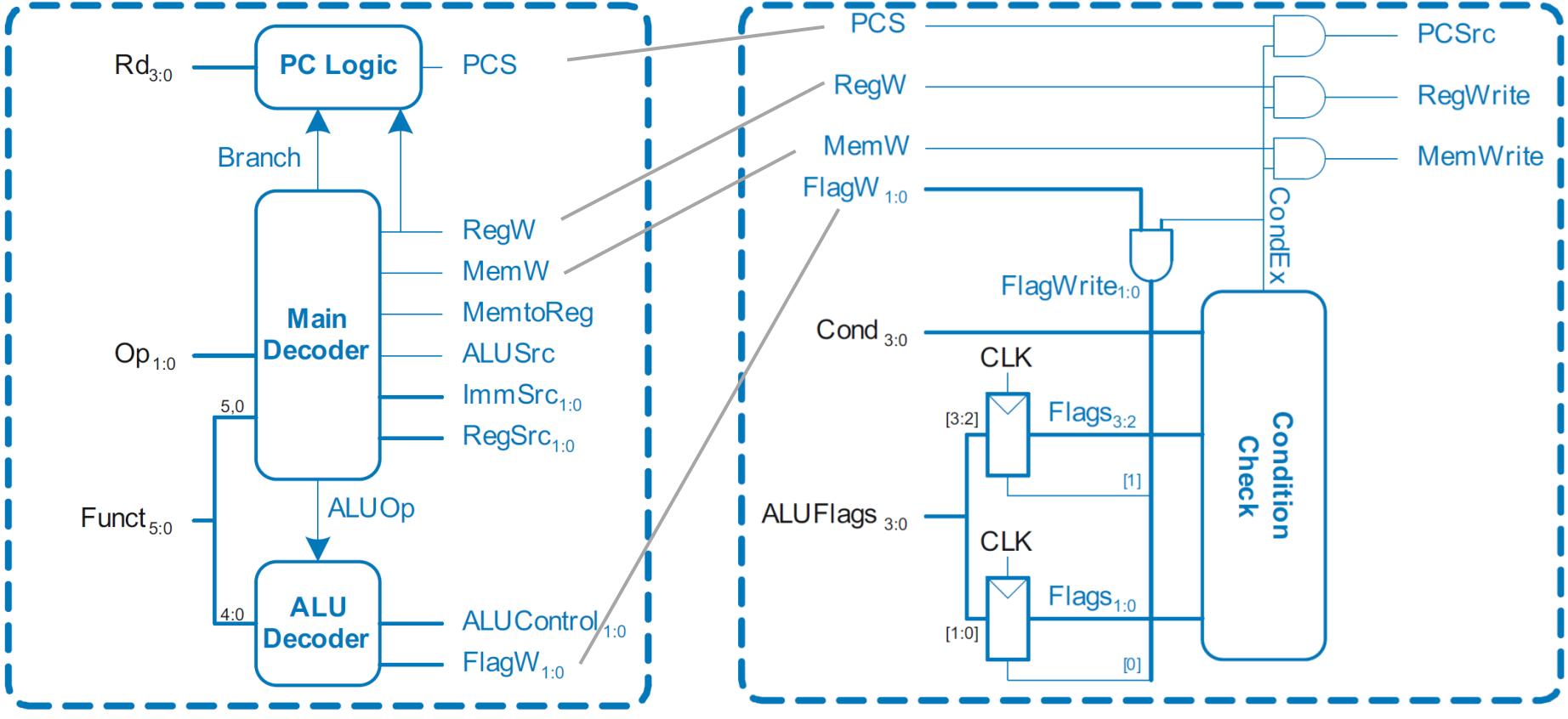
```
ORR Rd, Rn, Rm
```

Merk, ikke conditional.

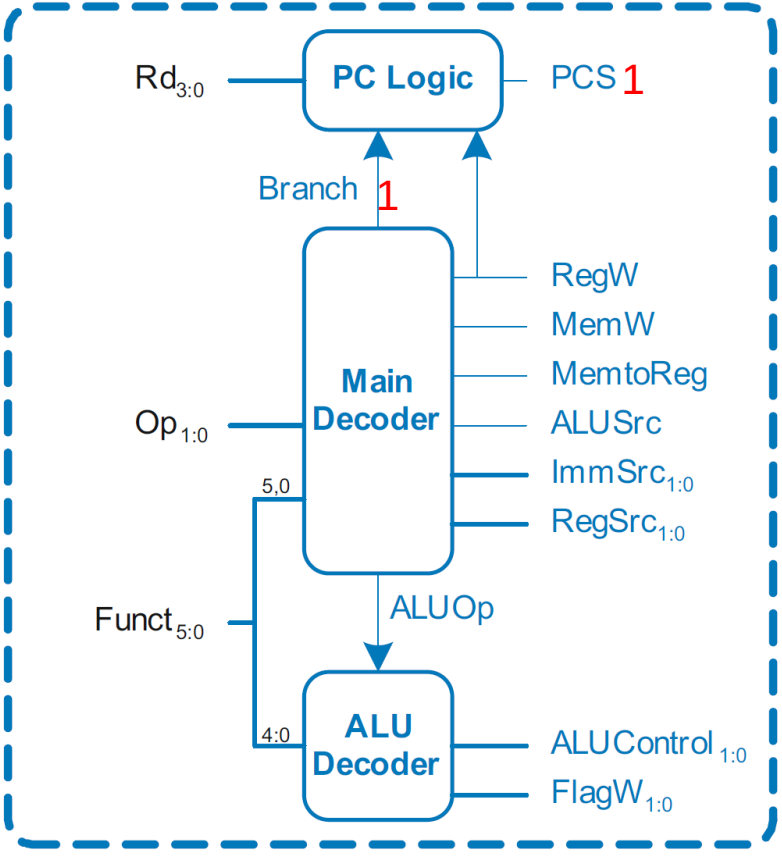


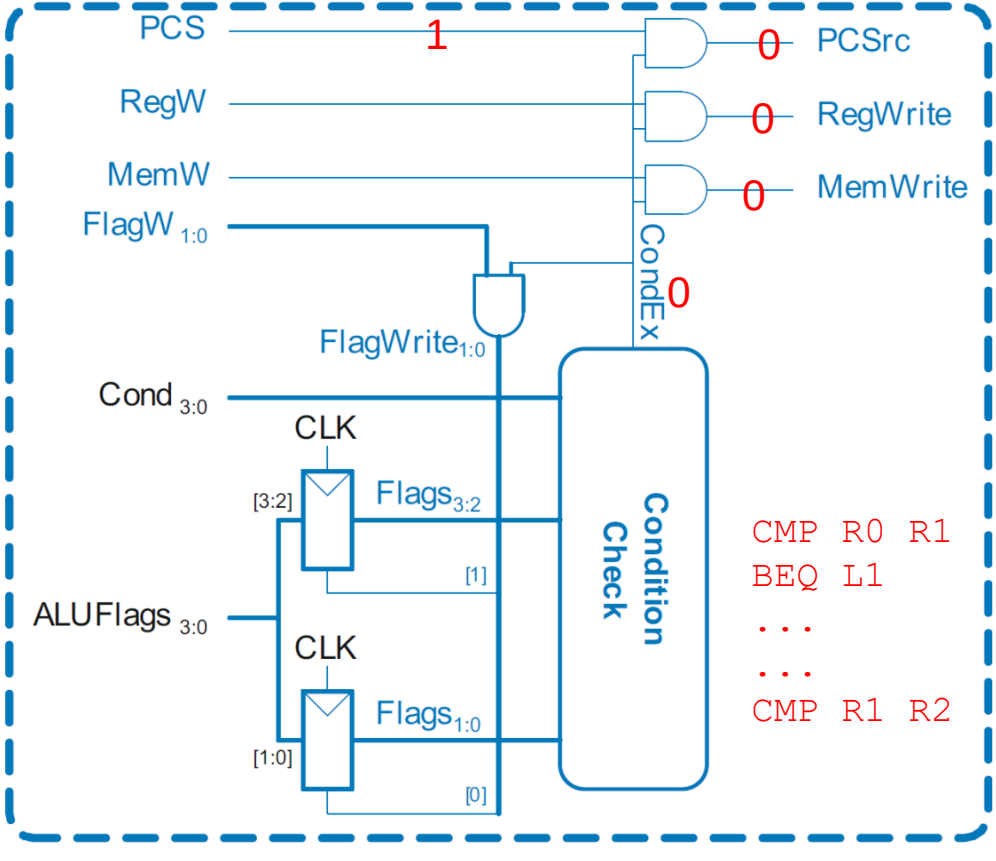
Det meste basert på kombinatorisk logikk.
Kun ALUFlags som går via flipflop-register.

Single-Cycle Control



Eksempel: BEQ (NZCV=0000)



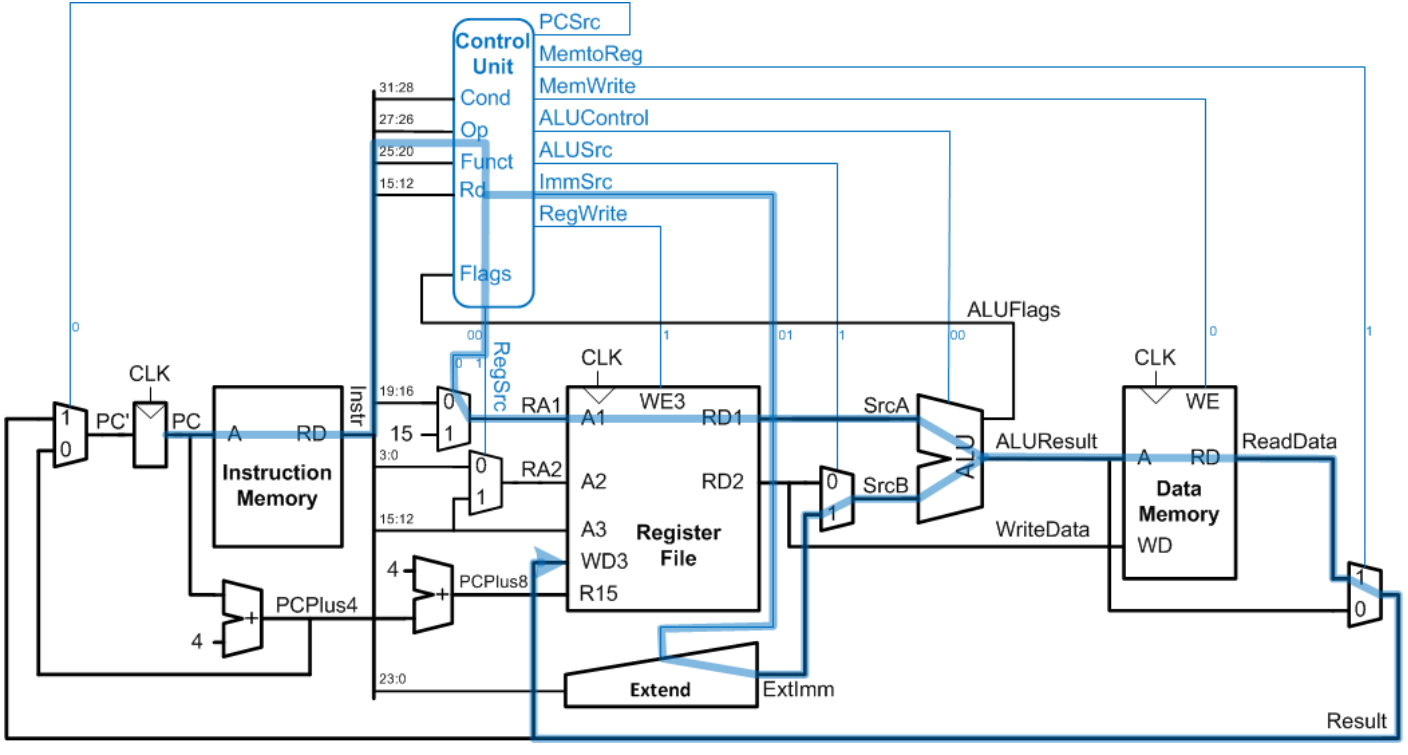


Single cycle prosessor - Ytelse

- *Program execution time*
  - **Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)**
- Definisjoner: CPI T_C
  - CPI: Cycles/instruction
  - clock period: seconds/cycle
  - IPC: instructions/cycle = IPC

Hvordan finner vi klokkeperioden (klokkehastigheten)?

Single-Cycle Performance: Maks klokkehastighet



T_c er begrenset av critical path (`LDR`)

Single-Cycle Performance Example

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
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Den instruksjonen som tar lengst tid bestemmer hvilken klokkehastigheten vi kan ha.

$$T_{c1} = t_{pcq_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$$
$$= [50 + 2(200) + 70 + 100 + 120 + 2(25) + 60] \text{ ps}$$
$$= \mathbf{840\ ps} \Rightarrow \mathbf{1.2}\ \text{Ghz}$$

Single-Cycle Performance Example

Et program med 100 milliarder instruksjoner:

Execution Time = # instructions x CPI x T_C

= (100×10^9)x (1) x $(840 \times 10^{-12}$ s)

= **84 sekunder**

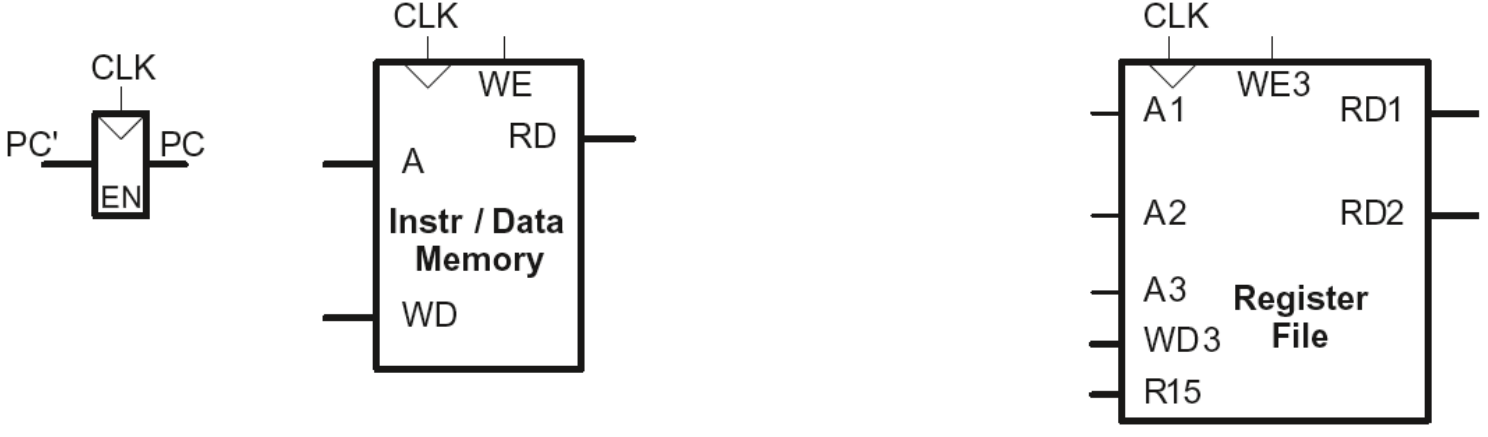
- Kan vi med et mer sofistikert design få bedre ytelse?
- Hvordan?

Multicycle ARM Processor

- **Single-cycle:**
  + enkel
  - Klokkehastighet begrenset av den tregeste instruksjonen (`LDR`)
  - Separate minne for instruksjoner og data
  - Trenger 2 ekstra addere
- **Multicycle:**
  + høyere klokkehastighet
  + enkle instruksjoner kan kjøre raskere
  + gjenbruker dyr hardware
  - Krever ekstra hardware som koster litt ekstra tid (overhead)

Multicycle tilstandselementer

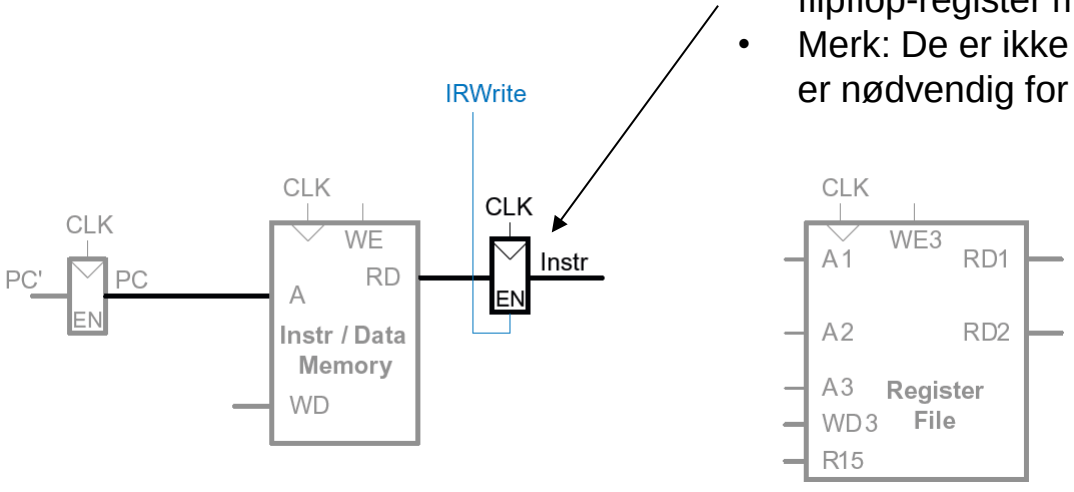
- Erstatter instruksjonsminne og dataminne med et felles minneelement (mer realistisk)

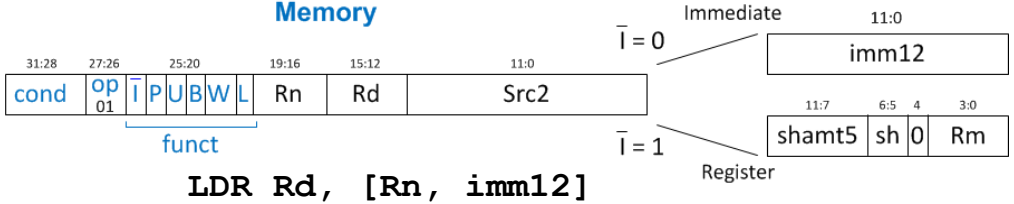


Multicycle Datapath: Instruction Fetch

- **Steg 1:** Hent instruksjon (fetch)

- For å få til multicycle datapath må vi legge til flipflop-register mellom elementene.
- Merk: De er ikke endel av arkitekturen, men er nødvendig for å få til funksjonaliteten.



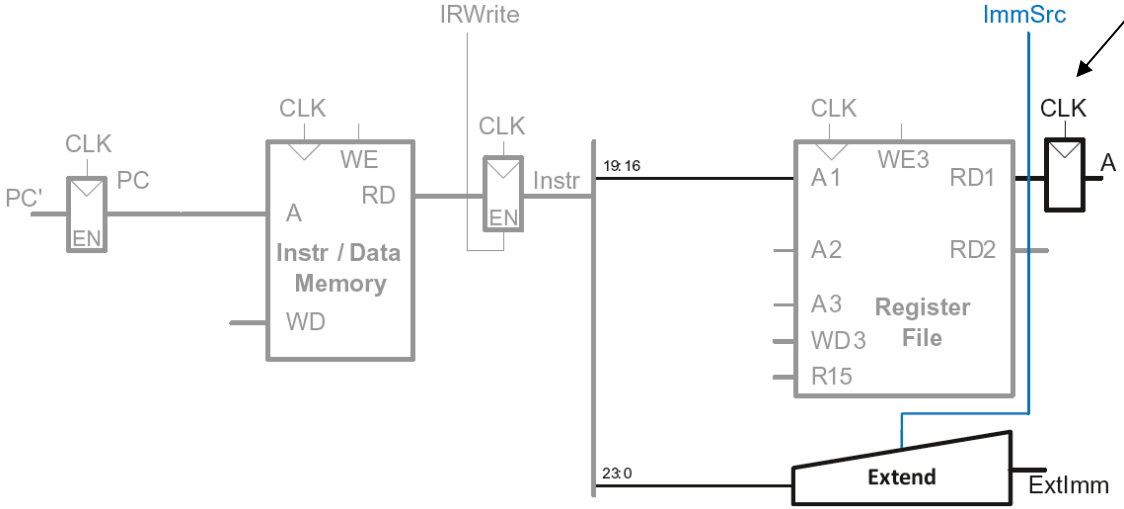


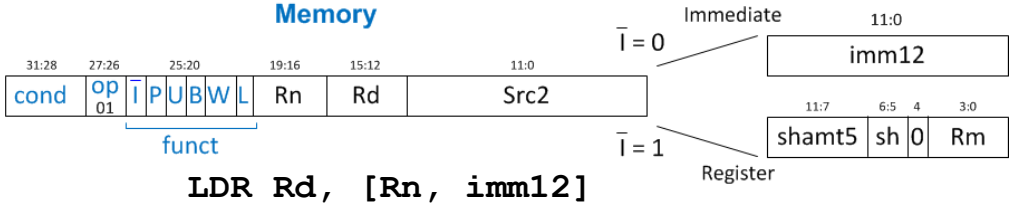

```
LDR Rd, [Rn, imm12]
```

Multicycle Datapath: LDR Register Read

- **STEP 2:** Hent kilde operander fra RF

Legger resultatet av register fetch på et nytt buffer.



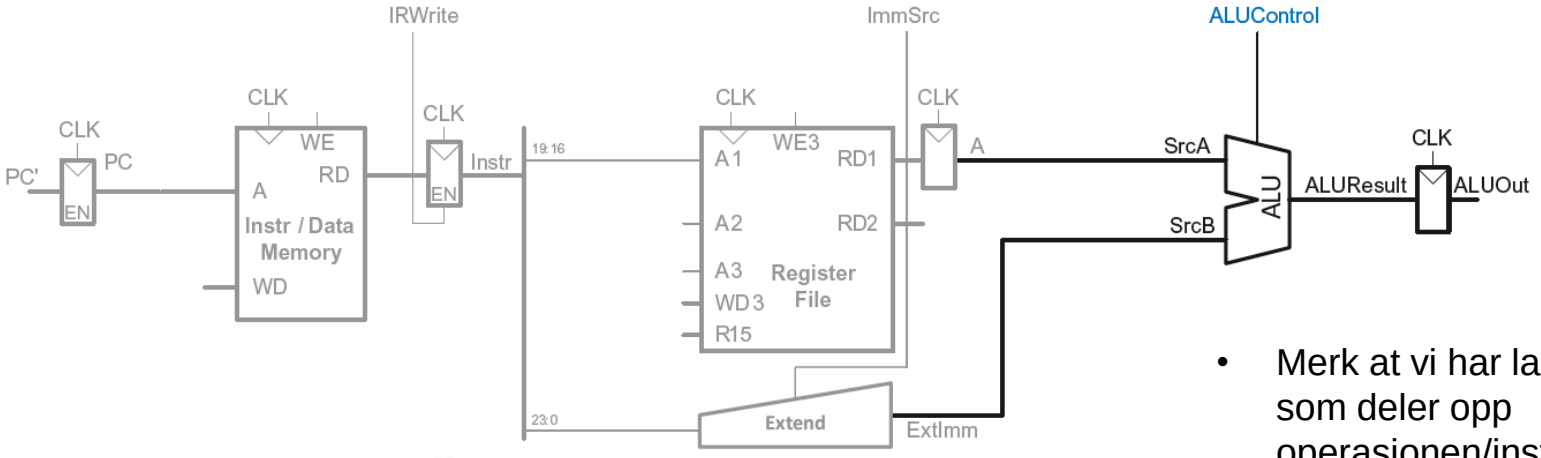


`LDR Rd, [Rn, imm12]`

Multicycle Datapath: LDR Address

- **Steg 3:** Beregn minneadressen

Vi kan utføre ALU operasjonen som før, og legge ALU resultatet på et nytt buffer.

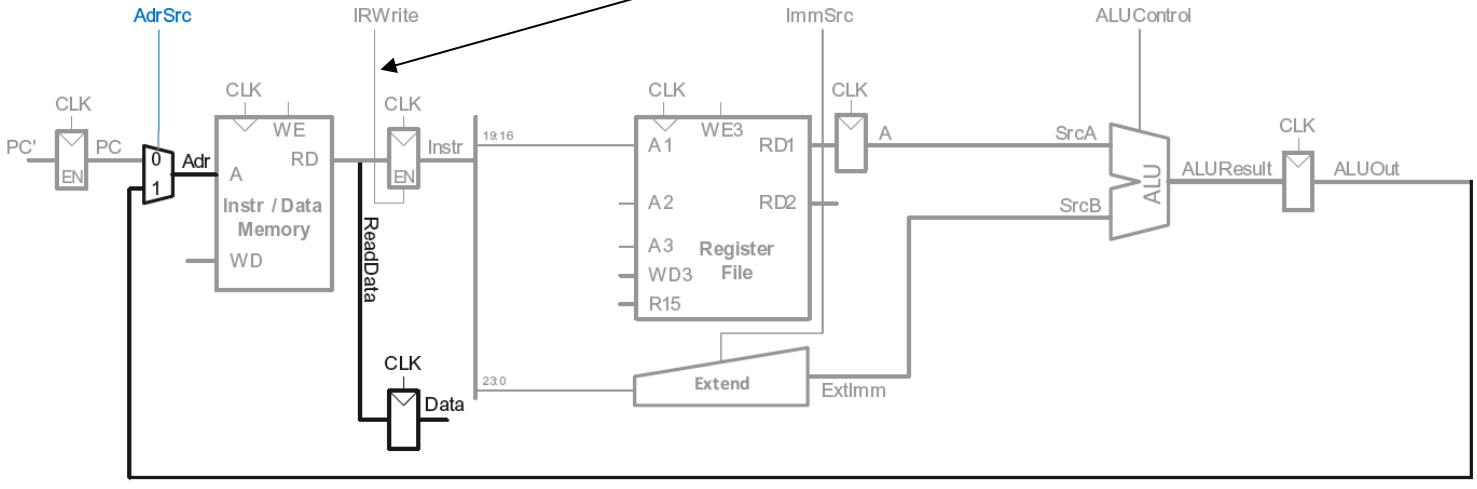


- Merk at vi har lagt til 3 buffere som deler opp operasjonen/instruksjonen.
- Instruksjonen vil derfor ta flere klokkesykler, men hver av delene går kjappere.

Multicycle Datapath: LDR Memory Read

- **Steg 4:** Les data fra minne

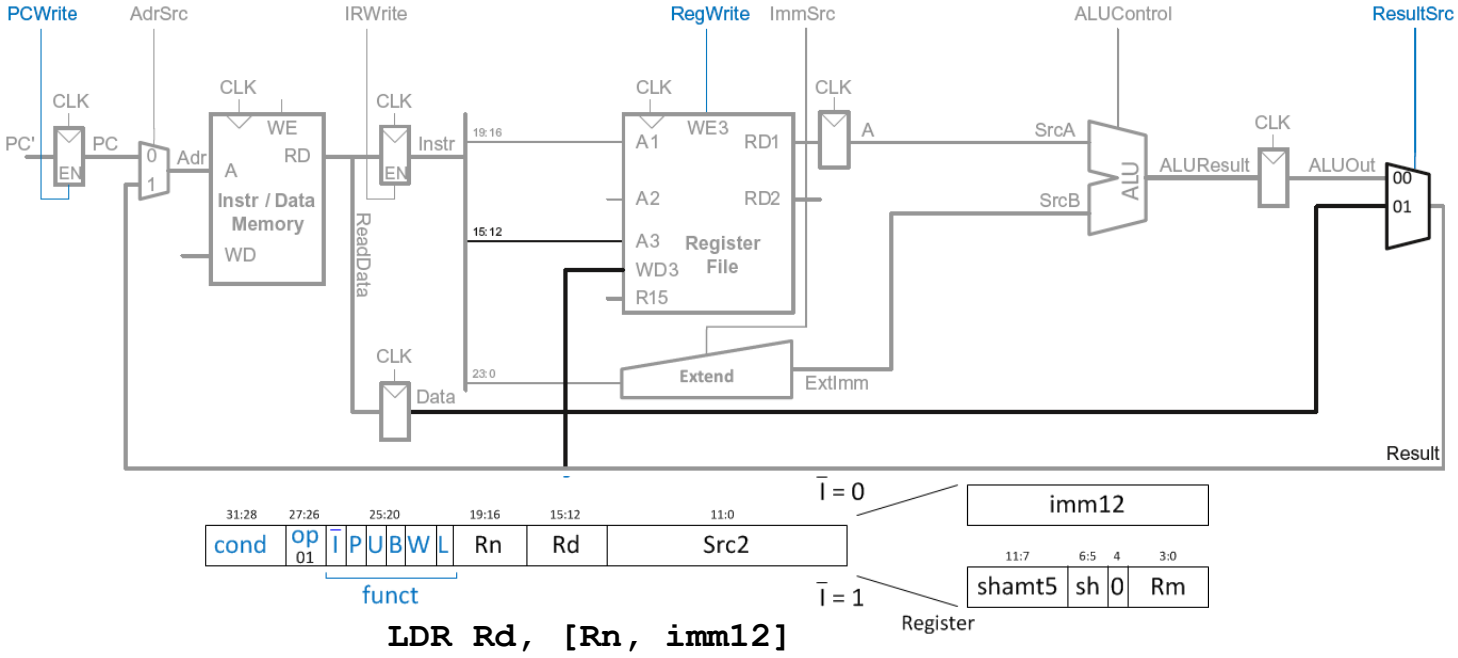
IRWrite sørger for at vi holder instruksjonen så lenge vi trenger den.



Merk at *Instr* spesifiserer hvilket register element vi skal skrive til. Så vi er avhengig av at *Instr* bit'ene låses så lenge vi trenger dem.

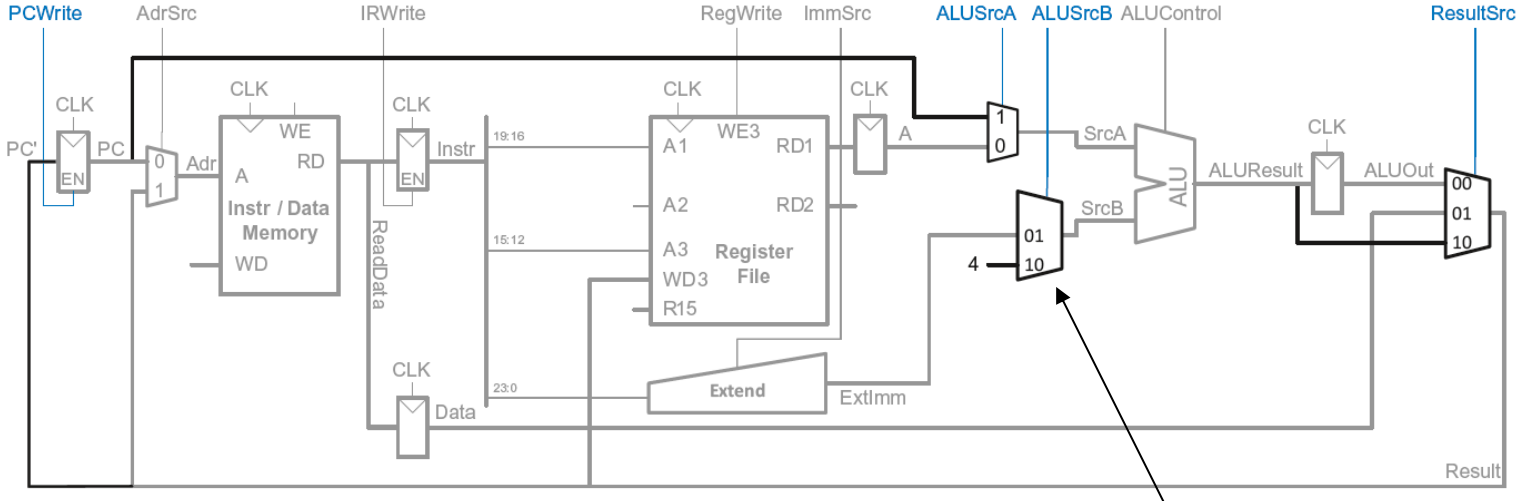
Multicycle Datapath: LDR Write Register

- **Steg 5:** Skriv data tilbake til register filen.



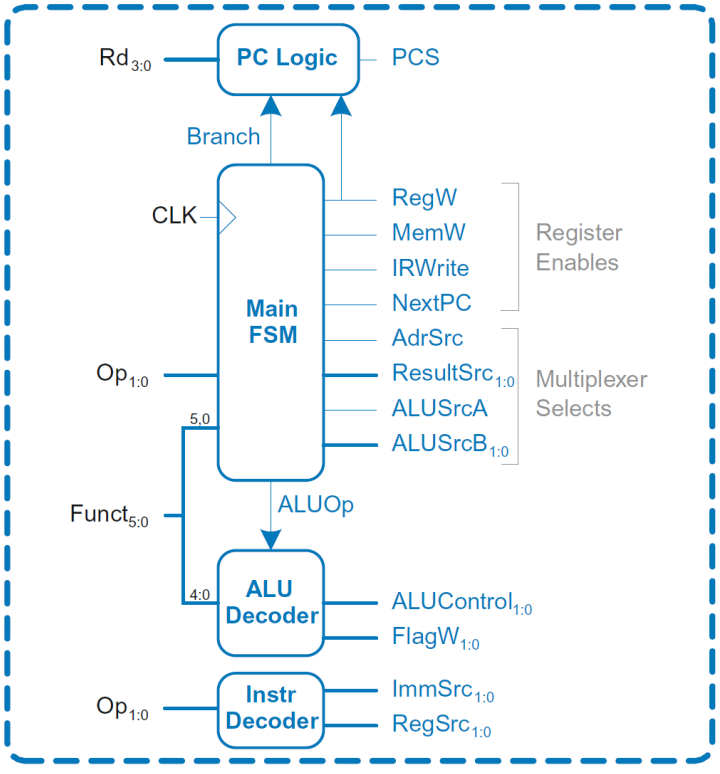
Multicycle Datapath: Increment PC

- **Steg 6:** Øk PC med 4



Gjenbruker ALU til å øke PC med 4.

Multicycle Control: Decoder

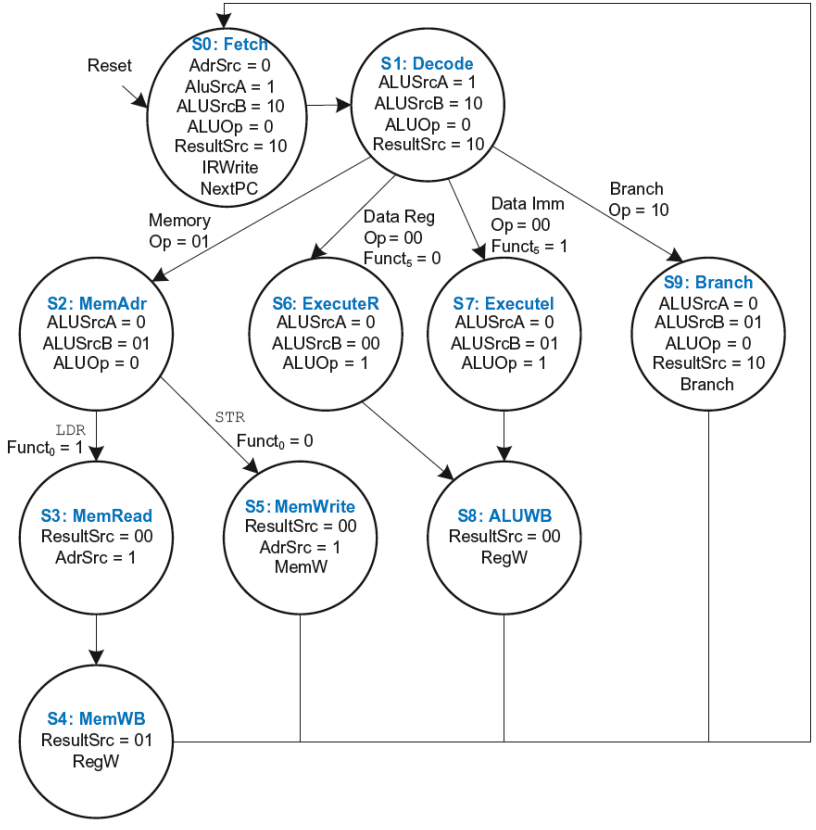


- Får mer komplisert *decoder*.
- Noe av tolkningen er fortsatt kombinatorisk logikk.
- Men mye må løses som en tilstandsmaskin.
  - Vi trenger forskjellige kontrollsignaler i de forskjellige stegene i instruksjonsgangen.

Bokens utledning av nødvendig tilstandsmaskin.

Multicycle Controller FSM

State	Datapath µOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC +4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
ExecuteI	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset



Legg merke til at forskjellige instruksjoner har forskjellige antall skritt

Multicycle Processor - Ytelse

- Instruksjonene har forskjellige antall skritt/cycles:
  - 3 cycles: `B`
  - 4 cycles: DP, `STR`
  - 5 cycles: `LDR`
- CPI blir et vektet gjennosnitt.
  - SPECINT2000 benchmark:
  - **25%** loads
  - **10%** stores
  - **13%** branches
  - **52%** R-type
- **Average CPI** = (**0.13**)(3) + (**0.52 + 0.10**)(4) + (**0.25**)(5) = **4.12**

Multicycle Performance Example

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
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Klokkehastigheten bestemmes av det skrittet som tar lengst tid (worst case)

$$T_{c2} = t_{pcq} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup}$$
$$= [40 + 2(25) + 200 + 50] \text{ ps} = \mathbf{340 \text{ ps} = > 2.94 \text{ Ghz}}$$

Multicycle ytelse

- For et program med **100 milliarder** instruksjoner utført på **multicycle** prosessoren
  - **CPI** = 4.12 cycles/instruction
  - **Klokkeperiode:** T_{c2} = 340 ps

- **Execution Time =** (# instructions) × CPI × T_c
  = $(100 \times 10^9) \times (4.12) \times (340 \times 10^{-12})$
  = **140 sekunder**

Tregere enn single-cycle processor (84 sekunder)

- Men bruker et felles minne og en ALU (mer kompakt design)

Moral: Høyere klokkehastighet gir ikke nødvendigvis høyere ytelse.

Oppsummering

- Vi har sett hvordan en prosessor kan
  - *Fetche* og tolke instruksjoner
  - Utføre forskjellige instruksjoner
  - Litt om kontrollsignaler
- Sett på to forskjellige mikroarkitekturer
  - Singel-cycle
  - Multi-cycle
- Neste gang
  - Pipeline
- **Ukeoppgaver**: 7.3*, 7.7 og 7.8.