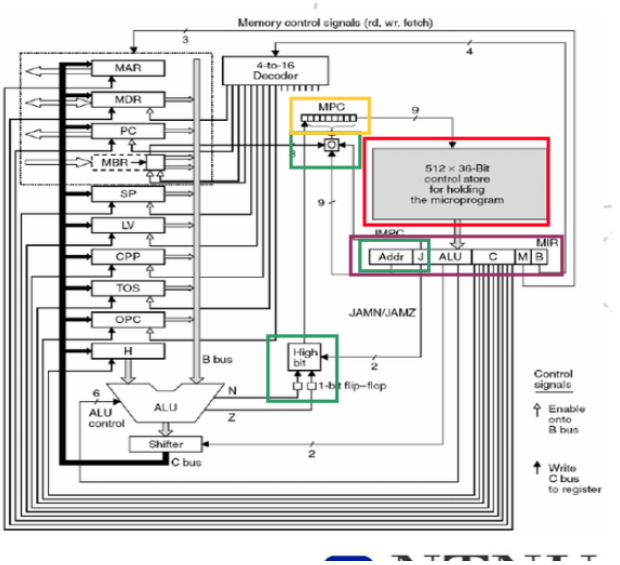
Week 11 MIC 1, 2, 3 og 4

Vi skal gå gjennom prosesser for å øke ytelsen til en prosessor. Med ytelse så mener vi hastighet.

MIC 1

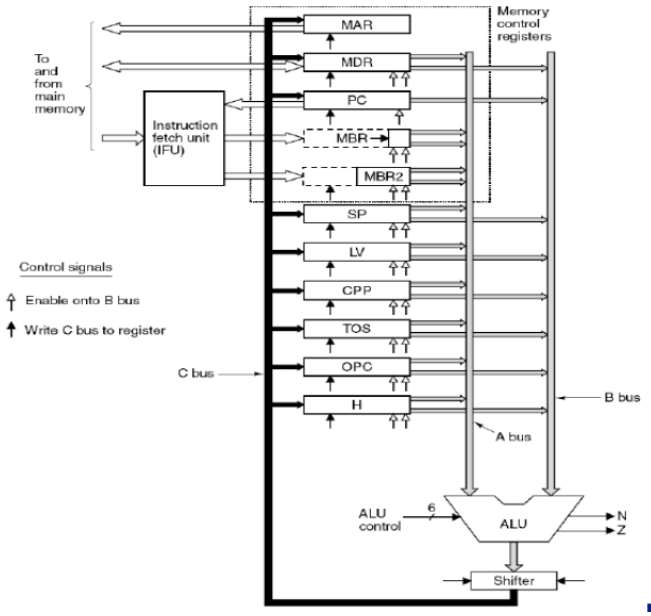
Dette har vi sett nøye på allerede:

- Hent instruksjon
  - PC peikar på instruksjon
  - Fetch controlsignal
  - Instruksjon i MBR
  - MBR -> MPC
- Decode
  - MPC peikar på adr. i control store
- Utfør instruksjon
  - Sekvens mikroinstruksjonar
    - Styresignal til data path (MIR)
    - Oppdater MPC
- Oppdater PC
  - PC <- adr. til neste instruksjon

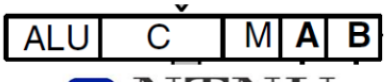


MIC 2

Vi får noen ekstra tillegg til arkitekturen:



- Innfører ein 3. buss, A-Buss
  - ALU tilgong til 2 register
    - Treng ikkje gå om H-register
  - Kan redusere antal mikroinstruksjonar
- Må innføre A-Buss felt I MIR
- Må endre og utvide control store
  - 4 ekstra bit, går frå 36 til 40 bit
- Ekstra busslinjer, auka areal
- Control store no 512 x 40-Bit
- MIR for data path med A- og B-buss:



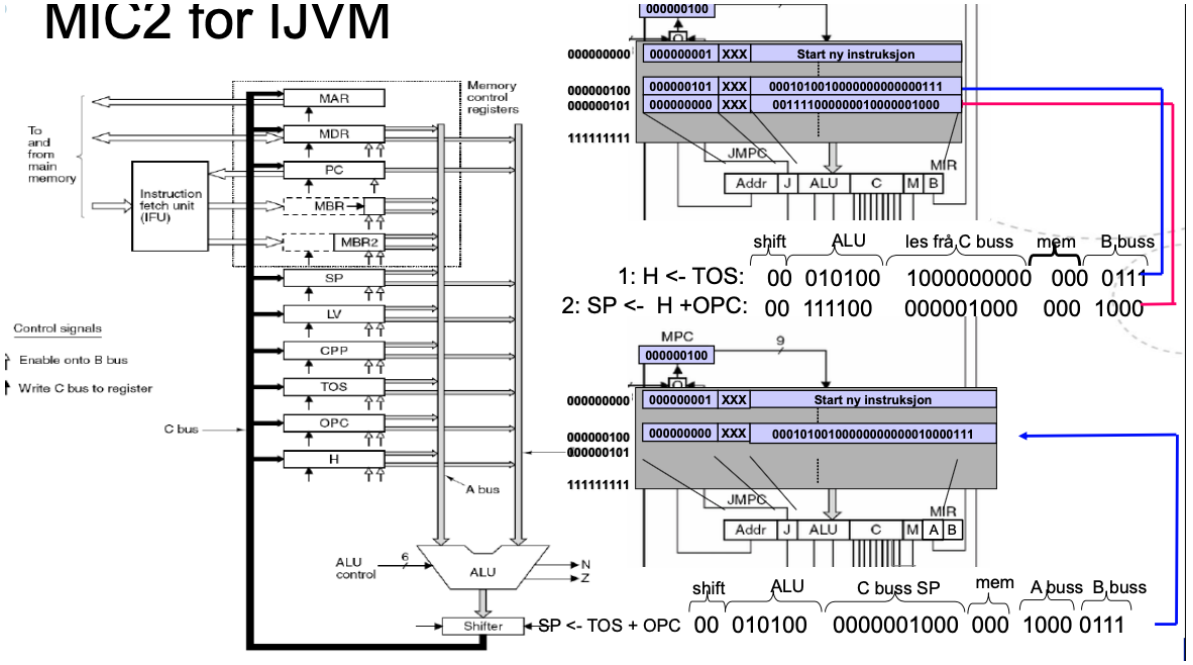
NTNU
Innovation and Creativity

Denne her har fått en boks som heter Instruction fetch unit (IFU) og den har fått en ekstra buss: A-bussen. Denne mikroarkitekturen kjører samme instruksjonssett som MIC 1. Dette er bare en annen implementasjon for å gjøre det samme instruksjonssettet.

ALU-en har plutselig tillgang til to register. Vi kan dermed redusere antall instruksjoner hvor vi må mellomlagre i H-registeret.

Endringene er at vi må innføre en A-buss felt i MIR og vi må utvide control store med 4 ekstra bit (fra 36 til 40)

Eksempel

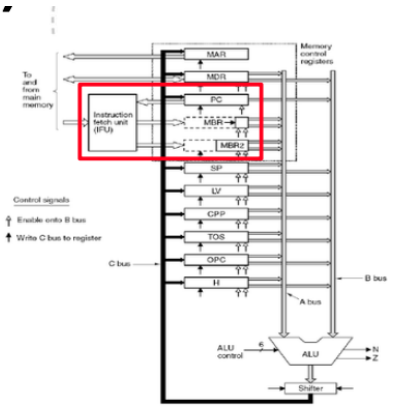


Over ser vi hva vi måtte gjøre tidligere og hva vi må gjøre nå. Nederst ser vi at vi kan legge sammen TOS og OPC i samme instruksjon. Dette er fordi vi kan velge både A- og B-buss samtidig, instruksjonen til ALU-en og hvilket register dette skal legges inn i. Vi har doblet ytelsen i dette eksempelet.

Instruction Fetch Unit (IFU)

Skal redusere klokkepulser som trengs for å hente instruksjonene våre. I MIC 1 må datapathen være med på å oppdatere PC.

- NO: For kvar instruksjon
  - PC increment (ALU)
  - PC peikar på neste instruksjon
  - Operandar lest frå minne
  - Operandar skrives til minne
  - ALU operasjon, resultat lagra (ein eller anna plass



- Også:
  - Instruksjonar med ekstra operandar
    - Kvar operand må hentast (1 Byte) Brukar PC dvs også ALU

- Kva viss
  - Ein kan sleppe å bruke datapath

- 1 ILOAD
- 2 PC = PC + 1
- 3 IADD
- 4 PC = PC + 1
- 5 ISTORE
- 6 PC = PC + 1
- 7 ILOAD
- 8 PC = PC + 1
- 9 BIPUSH
- 10 PC = PC + 1
- 11 IF_ICMPEQ (betinga hopp) **L1**
- 12 PC = Opdater PC
- 13 ILOAD
- 14 PC = PC + 1
- 15 BIPUSH
- 16 PC = PC + 1
- 17 ISUB
- 18 PC = PC + 1
- 19 ISTORE
- 20 PC = PC + 1
- 21 GOTO **L2**
- 22 PC = **L2**
- 23 BIPUSH **L1**
- 24 PC = PC + 1
- 25 ISORE
- 26 PC = PC + 1
- 27 XXXXX **L2**

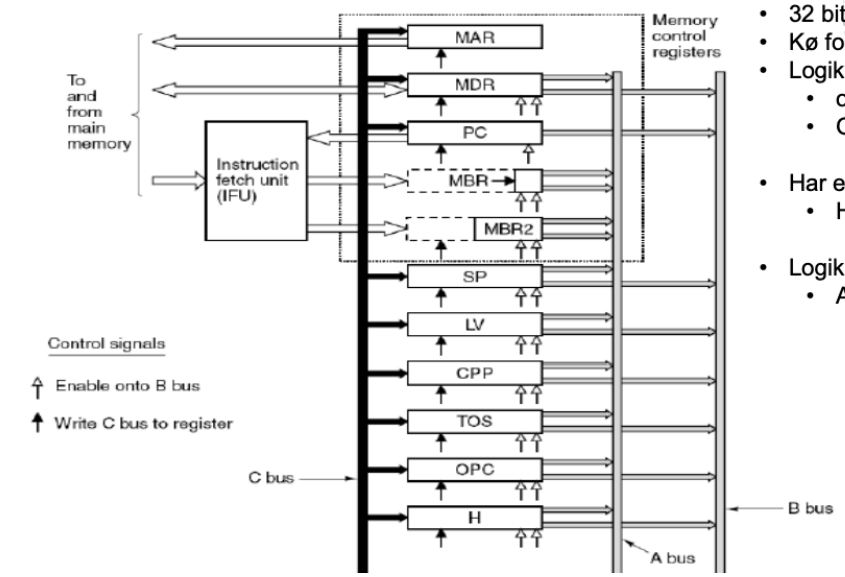
Hver gang vi oppdaterer PC må datapathen stå og gjøre control flyt operasjoner isteden for å regne.

Vi legger til en IFU som har en adderer som dermed kommer til å oppdatere PC. Vi må kun oppdatere PC ved en conditional branch:

- 1 ILOAD
- 2 IADD
- 3 ISTORE
- 4 ILOAD
- 5 BIPUSH
- 6 IF_ICMPEQ (betinga hopp) **L1**
- 7 PC = Opdater PC
- 8 ILOAD
- 9 BIPUSH
- 10 ISUB
- 11 ISTORE
- 12 GOTO **L2**
- 13 PC = **L2**
- 14 BIPUSH **L1**
- 15 ISORE
- 16 XXXXX **L2**

La oss se litt nærmere på hva som skjer her:

IFU instruksjon og operand kø



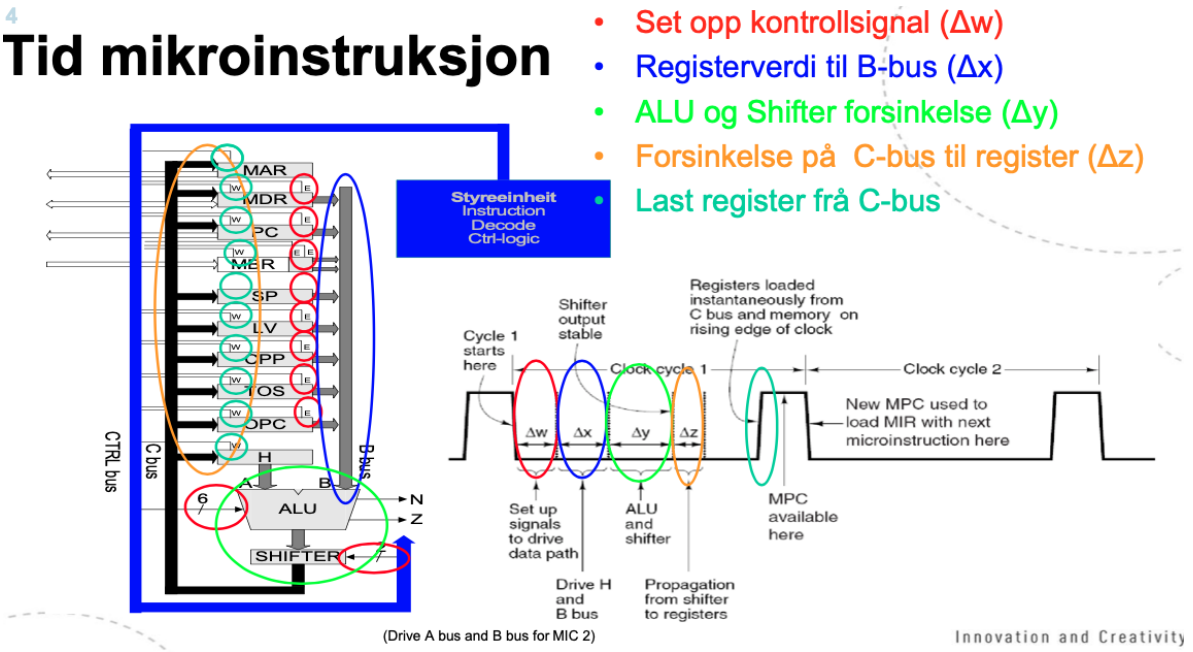
- IFU
- 32 bit buss mot prog minne
- Kø for instruksjonar og operandar
- Logikk for
  - operandar (MBR2)
  - OpCode (MBR)
- Har ein enkel “prefetcher”
  - Hentar opcode og Operandar
- Logikk for IFU
  - Auka areal
    - Adder
    - Shift register for kø
    - Logikk for styring
      - FSM/mikroOperasjonar
    - Logikk for decoding
      - opcode/Operand

NTNU
Innovation and Creativity

IFU-en styrer oppdateringen av PC slik at vi slipper å bruke datapath på dette. Den har innført en kø hvor vi endrer på selve minnegrensesnittet som er en 32-bit buss til programminnet. Basert på om bytes er en operand eller instruksjon så vil dette bli henholdsvis lagt til i MBR2 eller MBR. Dermed har vi flyttet en del av dekodingslogikken til IFU.

MIC 3

Vi korter ned på klokkeperioden for å få til flere instruksjoner per sekund. Hva bruker vi tiden på?

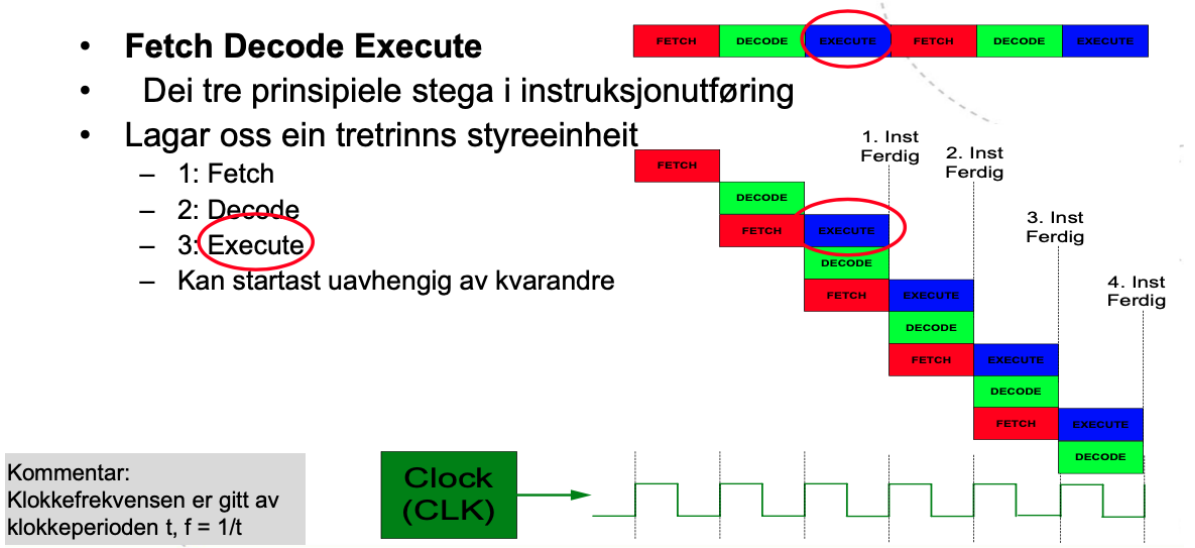


- Rød (Δw)
  - Vi setter opp kontrollsignalene våre først. Dette er EN signalene til registrene, setter opp ALU og shifter.
- Blå (Δx)
  - Det tar litt tid å flytte data fra register til bussene våre
- Grønn (Δy)
  - Tar tid for de stabile signalene i bussene å propagere gjennom ALU-en og shifter. Dette bruker mest tid.
- Orange (Δz)
  - Tiden fra vi har et stabilt signal fra shifteren til alle bussene våre er drevne. Altså at alle registrene har et gyldig signal inn.
- Turkis

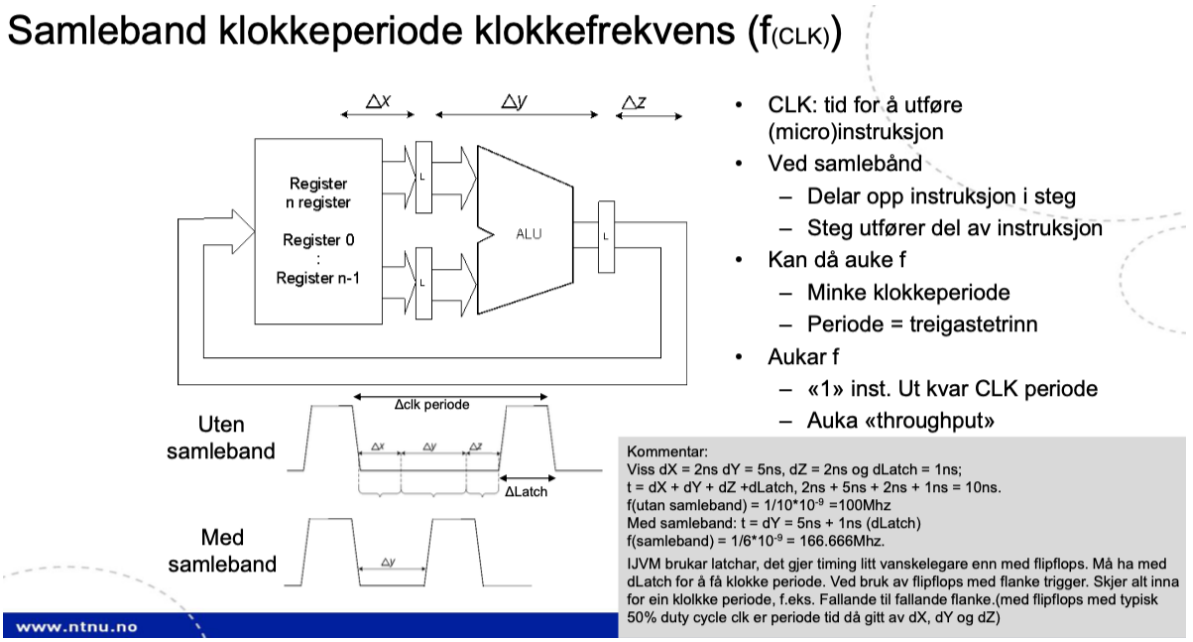
- Når klokkesignalet kommer så laser vi inn registrene fra buss

Vi skal bruke **instruksjonsnivåparallellitet (ILP)** fro å utføre flere instruksjoner samtidig (1 prosessor). Dette har vi snakket om tidligere.

Pipelining

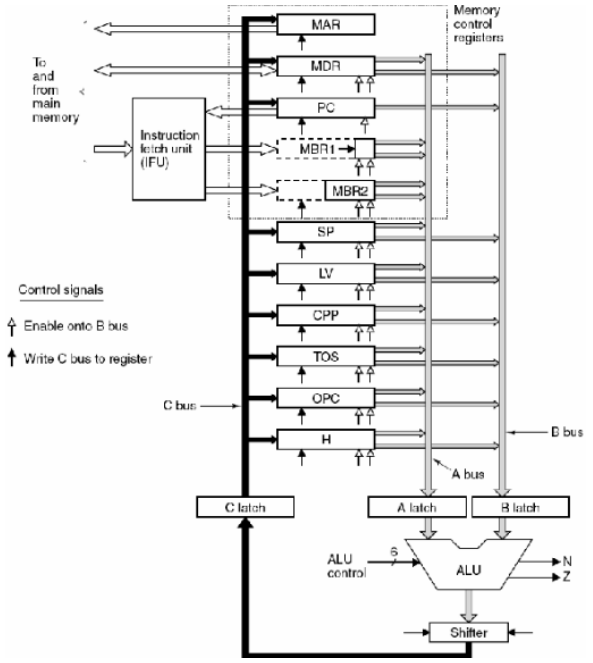


Vi har fetch, decode og execute. Vi skal se på execute. Det er denne deles som blir gjort i datapathen vår



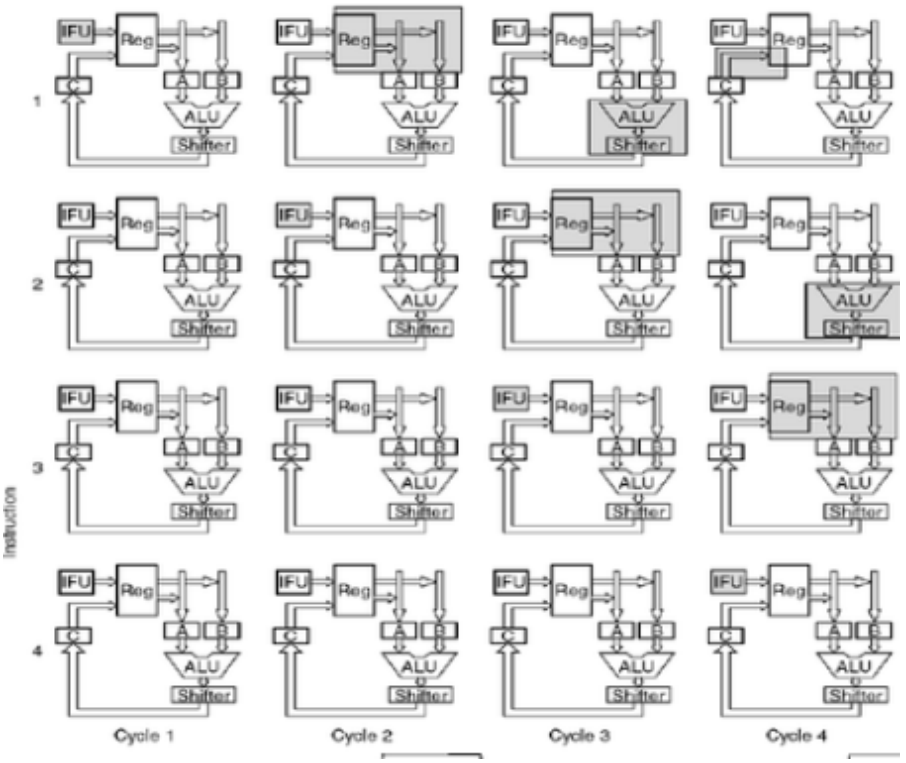
Vi kan tegne om en datapath til noe slik. Vi har Δx, Δy, og Δz. Δx er en tid for å få gyldig data ut av registrene våre, Δy er tid gjennom ALU og Δz er tiden til vi har gyldige verdier i bussen til registrene.

Dette betyr at disse instruksjonene kan gjøres samtidig. For å få til dette trenger vi latches (mellom register og ALU, samt etter ALU). Latchen slipper gjennom signal når klokken er høy. Ellers når klokken er lav slipper den ikke gjennom noe.
Først kan vi hente registrene våre og latche dem gjennom til ALU-en. F.eks så henter vi R0 og R5 som går gjennom latchen og inn i ALU-en. Etter det så er latchen stengt. Det vil si at vi kan hente inn R1 og R3 og putte det inn til latchen mens ALU-en utfører en operasjon og setter det inn i latchen utenfor. Neste klokkeperiode vil R1 og R3 gå inn i ALU mens R0 og R5 er i C-bussen.
Nå som vi kan gjøre alle Δ operasjonene samtid så trenger vi faktisk ikke å ha like lang klokketid. Vi kan kutte ned på klokketiden til den lengste mulige operasjonen. Siden Δy med ALU bruker lengst tid så kan klokkefrekvensen være Δy.
IJVM bruker lathcer noe som gjør timing litt vanskeligere enn med flip-flops. Vi må ha med dLatch for å få klokkeperiode.



På IJVM så tegnes det på følgende måte. Vi har registerbanken som tidligere. Vi har bussene våre som har latcher på utgangen, samt en latch ut av shifteren.

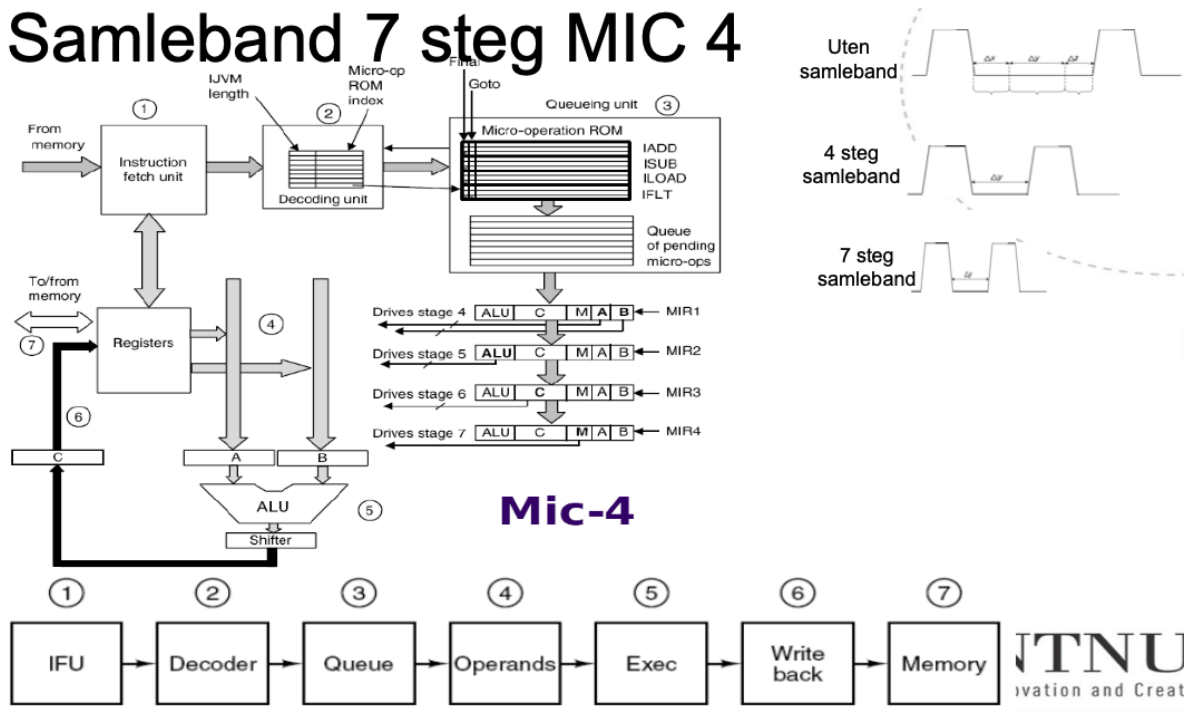
Vi ser også hvordan en 1-bit latch ser ut og funker nederst.



Her demonstrerer vi pipeliningen. Vi har instruksjon 1 til 4. Instruksjon 1 henter instruksjonene sine via IFU. På det tidspunktet på alt vente. Når instruksjon 1 putter registrene sine på bussen så kan instruksjon 2 starte med å hente sine instruksjoner via IFU. Når instruksjon 1 bruker ALU-en kan instruksjon 2 sette sine registre på bussene mens instruksjon 3 kan hente sine instruksjoner fra IFU. Dette fortsetter.

MIC 4

Vi kan ta MIC 4 et steg videre. Vi utvider med 7 samlebånd (pipelineing) steg.



Helt nederst så ser vi at vi har 7 forskjellige trinn isteden for 3.

Det er generelt vist hvilke ekstra komponenter vi trenger for å få til denne strukturen. Først ser vi at vi har MIR 1, 2, 3 og 4. Vi trenger en MIR for å legge ut signal til A og B- bussen, et for å drive ALU-en, en for å drive tilbakeskriving til register og en til å drive eventuel write til det eksterne minnet. Resultatet er at vi øker areal og har større effektforbruk. Siden vi også øker klokkefrekvensen så øker vi samtidig energiforbruket.