Pipelining

An Instruction Assembly Line

EEC170
Computer Architecture

FQ 2005

Courtesy of Prof. Kent Wilken and Prof. John Owens

Processor Designs So Far

- We have Single Cycle Design with low CPI but high CCT
- We have Multicycle Design with low CCT but high CPI
- We want best of both: low CCT and low CPI
- Achieved using pipelining

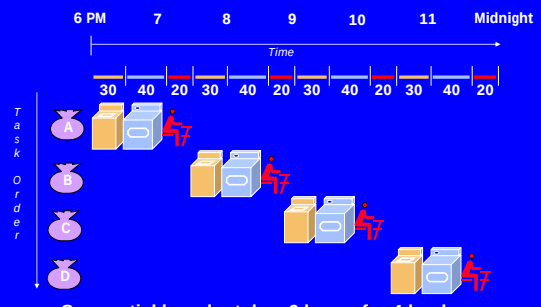
Pipelining is Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes

Laundry Timing

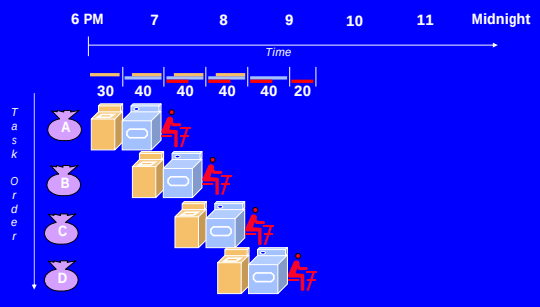
- How long does laundry take with a "single cycle" (wash/dry/fold is one clock) design? What is the clock cycle time? "CPI"? How many clocks?
- How long does laundry take with a "multiple cycle" (longest of wash/dry/fold is one clock) design? What is the CCT? "CPI"? How many clocks?

Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

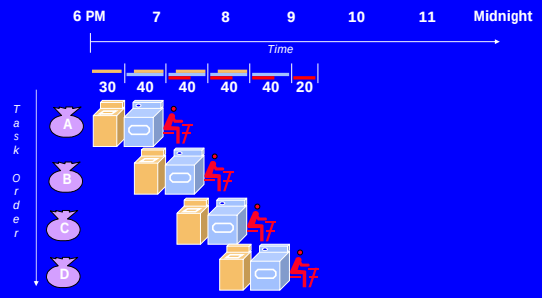
Pipelined Laundry: Start work ASAP



Laundry Timing

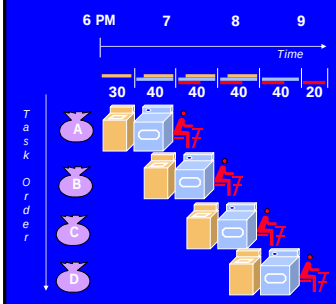
- How long does laundry take with a "pipelined" (longest of wash/dry/fold is one clock) design? What is the CCT? How many clocks?

Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



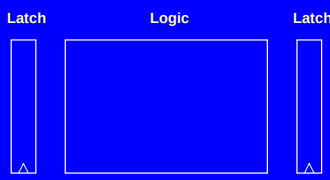
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

Sandwich Bar Analogy

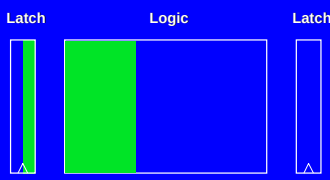
- Pipelining: Multiple people going through the sandwich bar at the same time
  - If you're in front of the pickles, but you don't need the pickles …
- Car assembly line

Digital System Efficiency

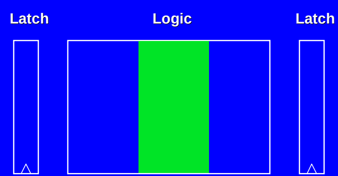
- A synchronous digital system doing too much work between clocks can be inefficient because logic can be static for much of clock period



Digital System Efficiency



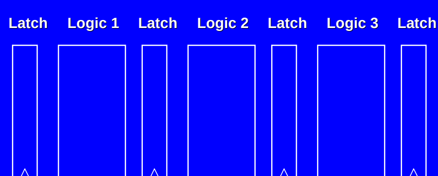
Digital System Efficiency



Digital System Efficiency

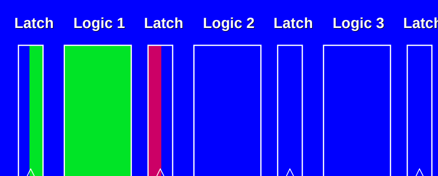
Pipeline Efficiency

- Efficiency can be improved by
  - I. subdividing logic into multiple stages and shortening CCT



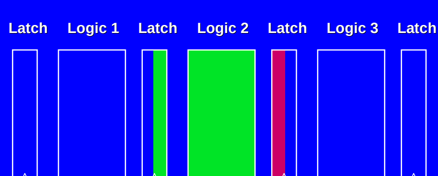
Single Pipelined Instruction

- Efficiency executing single instruction is even less efficient, why?



Single Pipelined Instruction

- Efficiency executing single instruction is even less efficient, why?

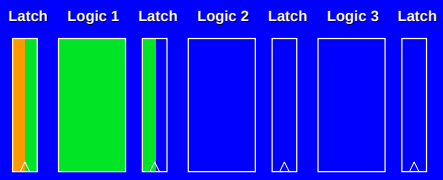


Single Pipelined Instruction

- Efficiency executing single instruction is even less efficient, why?

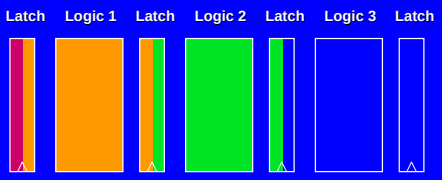
Pipeline Efficiency

- Efficiency can be improved by
  - I. subdividing logic into multiple stages and shortening CCT
  - II. overlapping operation execution



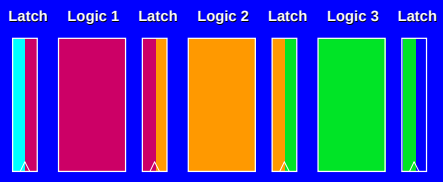
Pipeline Efficiency

- And by II. overlapping operation execution



Pipeline Efficiency

- And by II. overlapping operation execution



Pipeline Depth

- If three pipeline stages are better than non-pipelined, are four stages better than three?
- What is the limit to the number of stages?
- What is the CCT at that limit?

Why Pipeline?

- Suppose we execute 100 instructions. How long on each architecture?
- Single Cycle Machine
  - 4.5 ns/cycle, CPI=1
- Multicycle Machine
  - 1.0 ns/cycle, CPI=4.1
- Ideal pipelined machine
  - 1.0 ns/cycle, CPI=1 (but remember fill cost!)

Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 4.5 ns/cycle x 1 CPI x 100 inst = 450 ns
- Multicycle Machine
  - 1.0 ns/cycle x 4.1 CPI (due to inst mix) x 100 inst = 410 ns
- Ideal pipelined machine
  - 1.0 ns/cycle x (1 CPI x 100 inst + 4 cycle fill) = 104 ns

Fill Costs

- Suppose we pipeline different numbers of instructions. What is the overhead of pipelining?
- Ideal pipelined machine
  - 1.0 ns/cycle, CPI=1, 10 instructions
  - 1.0 ns/cycle, CPI=1, 1000 instructions
  - 1.0 ns/cycle, CPI=1, 100,000 instructions

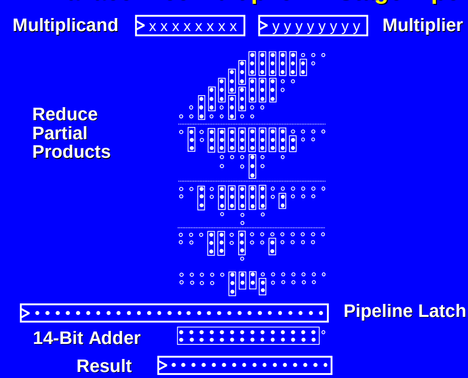
Fill Costs

- Overhead: Ideal pipelined machine
  - 1.0 ns/cycle, CPI=1, 10 instructions
  - 10 ns runtime + 4 ns overhead: 40% overhead
  - 1.0 ns/cycle, CPI=1, 1000 instructions
  - 1000 ns runtime + 4 ns overhead: 0.4% overhead
  - 1.0 ns/cycle, CPI=1, 100,000 instructions
  - 100,000 ns runtime + 4 ns overhead: 0.004% overhead

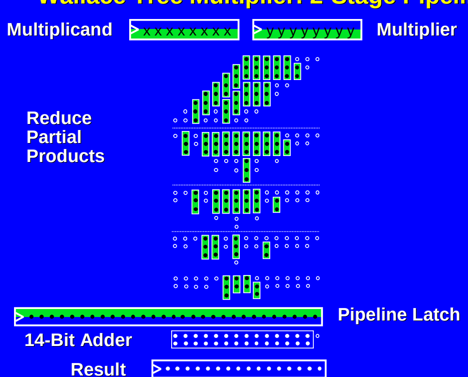
Pipelined Multiplier

- We can take the Wallace tree multiplier we developed in Chapter 3 and create a two stage pipeline
  - Reduce partial product terms
  - Do final n-2 bit addition using carry look-ahead adder
- The two operations are roughly balanced, both take log(n) time

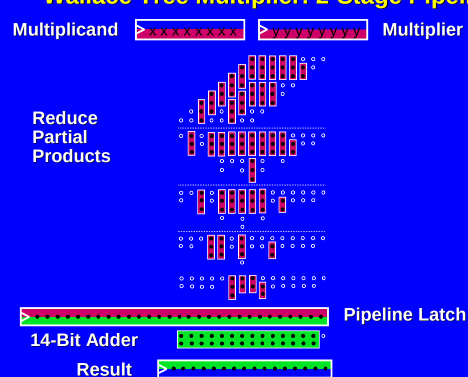
Wallace Tree Multiplier: 2-Stage Pipeline



Wallace Tree Multiplier: 2-Stage Pipeline



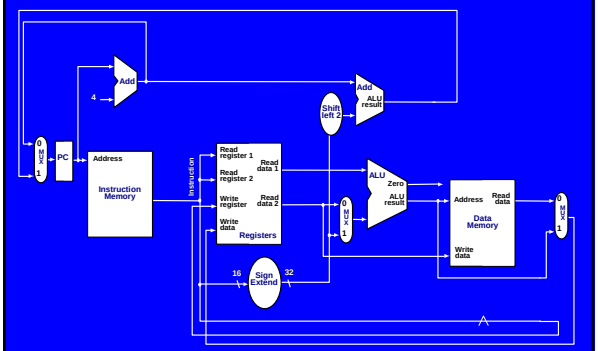
Wallace Tree Multiplier: 2-Stage Pipeline



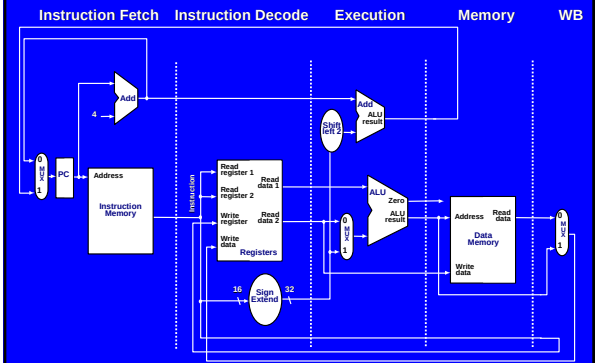
Pipelined Processor Design

- We'll take the single-cycle design and transform it into a pipelined design
- What we formally called a 'phase' of an instruction's execution will now become a pipeline stage

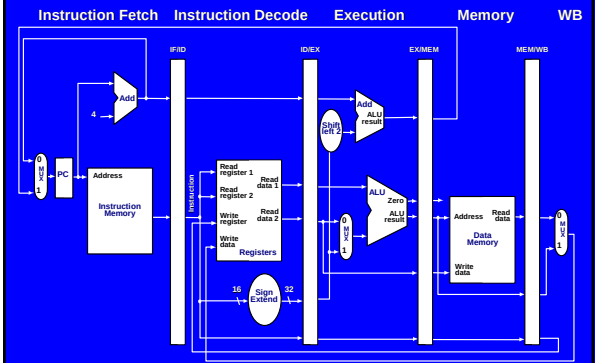
Single-Cycle Design



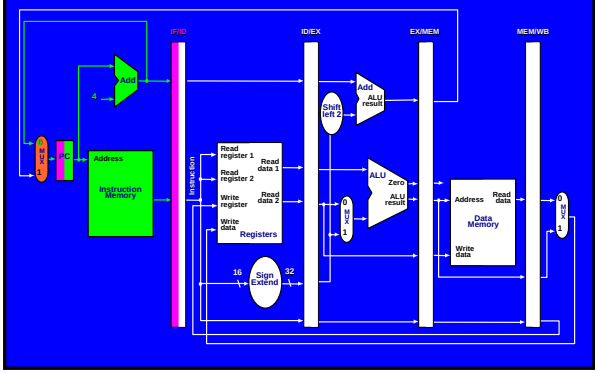
Single-Cycle Design Phases



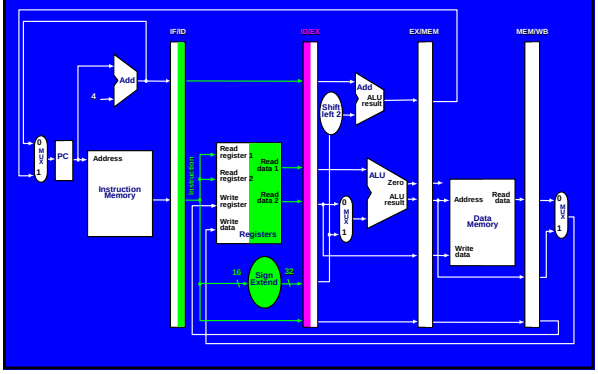
Pipelined Design



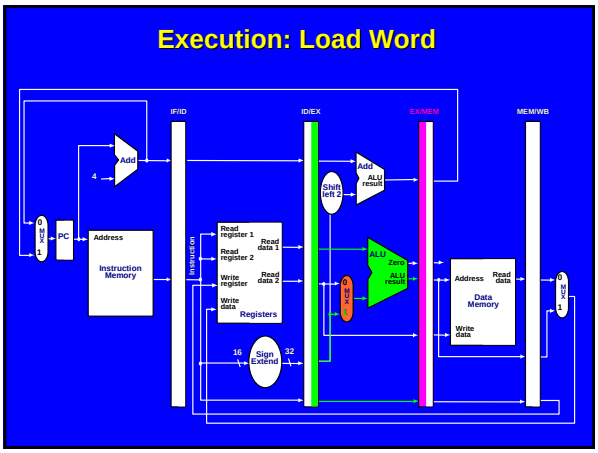
Instruction Fetch & PC + 4



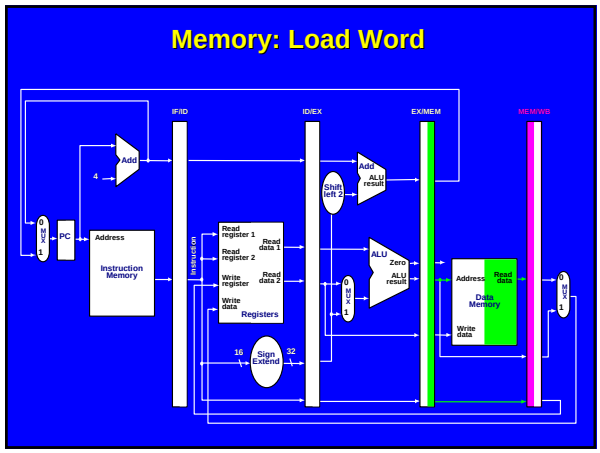
Instruction Decode & Reg Read



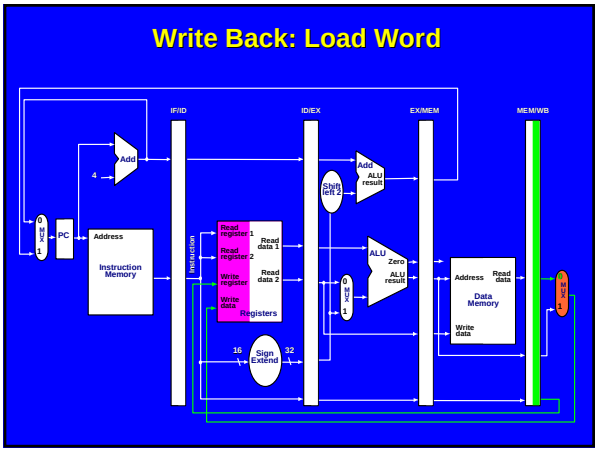
Execution: Load Word



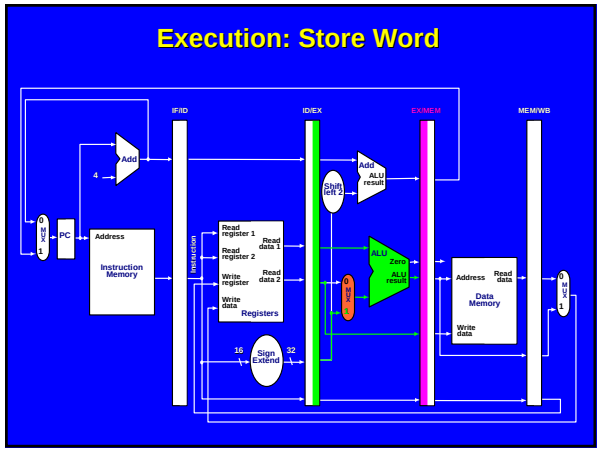
Memory: Load Word



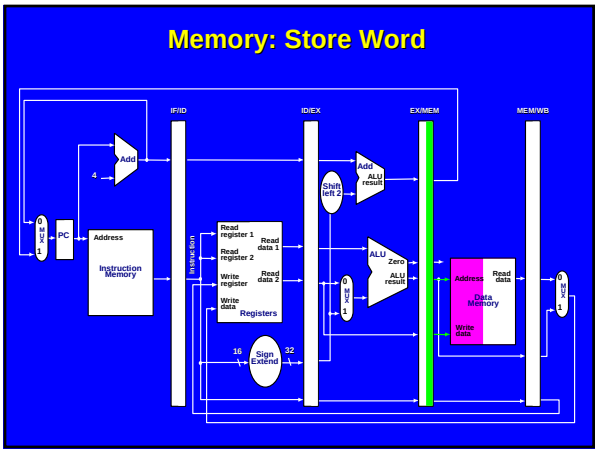
Write Back: Load Word



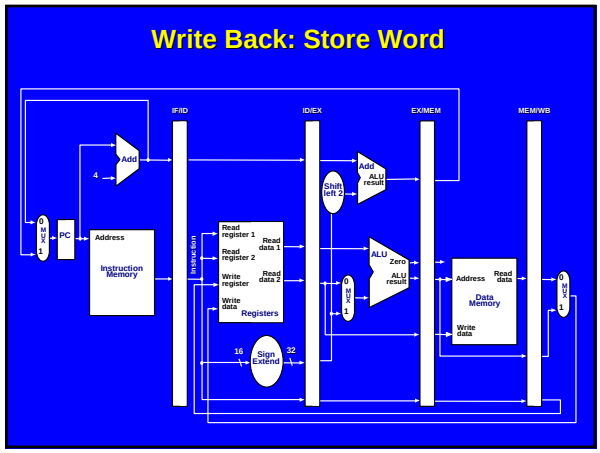
Execution: Store Word

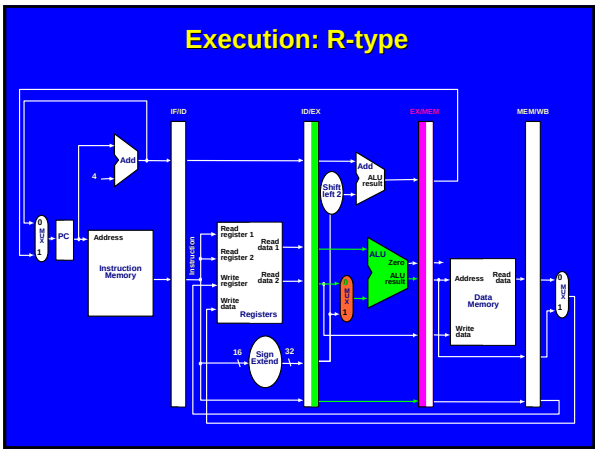


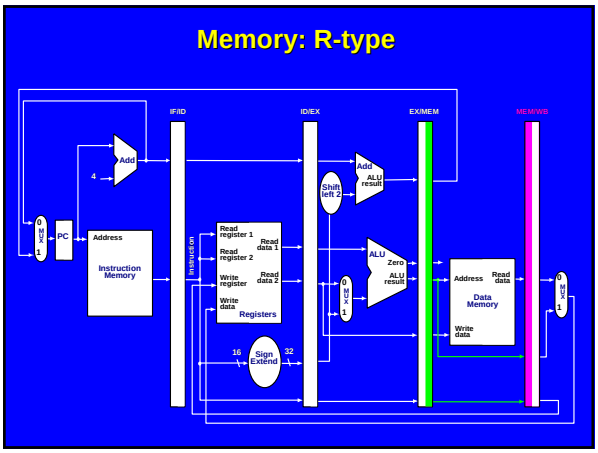
Memory: Store Word

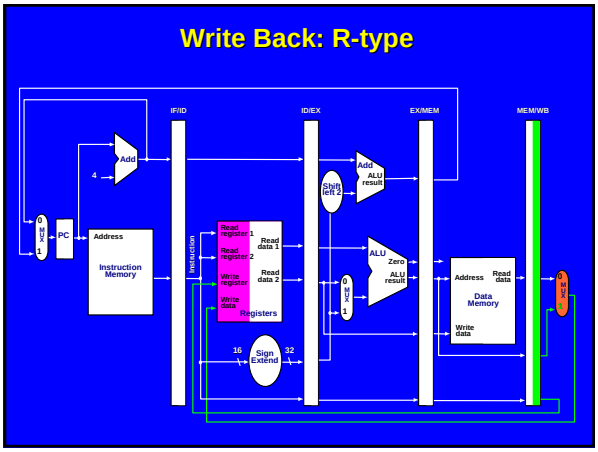


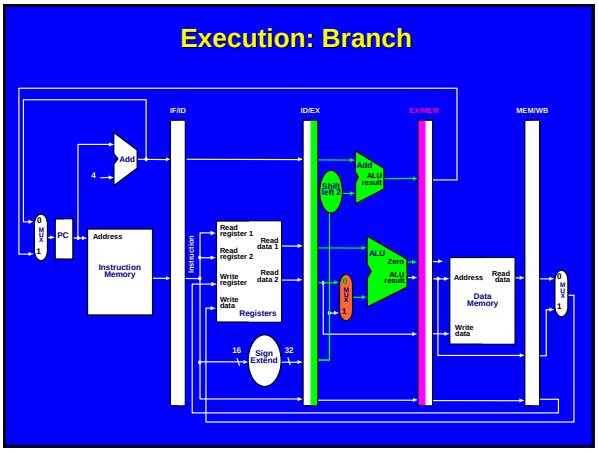
Write Back: Store Word

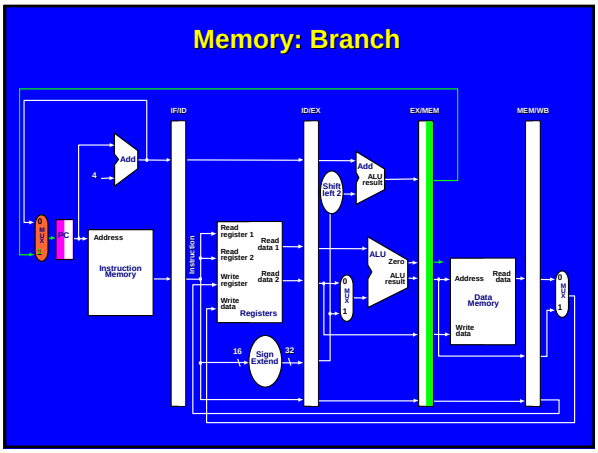


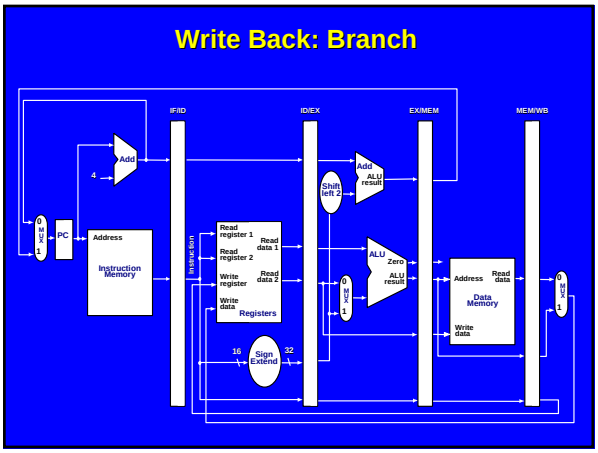

Execution: R-type
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction Memory
Instruction
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Shift left 2
Add
ALU result
ALU
Zero
ALU result
16
Sign Extend
32
Address
Read data
Data Memory
Write data

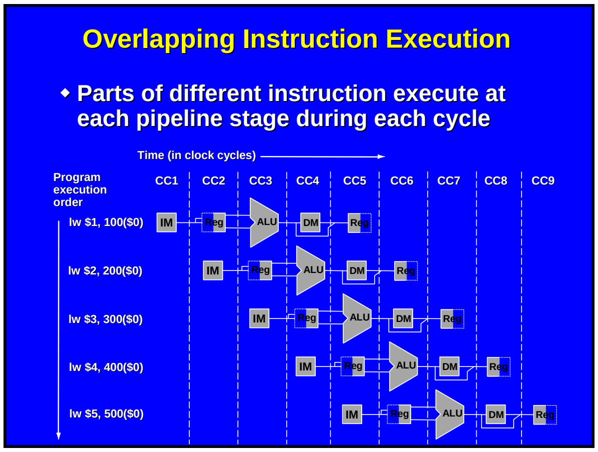

Memory: R-type
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction Memory
Instruction
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Shift left 2
Add
ALU result
ALU
Zero
ALU result
16
Sign Extend
32
Address
Read data
Data Memory
Write data

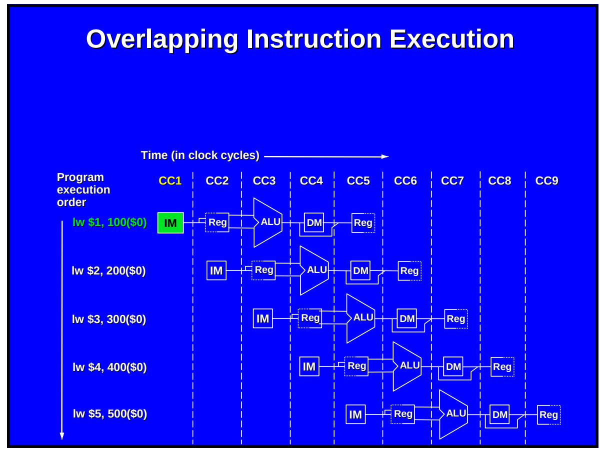

Write Back: R-type
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction Memory
Instruction
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Shift left 2
Add
ALU result
ALU
Zero
ALU result
16
Sign Extend
32
Address
Read data
Data Memory
Write data

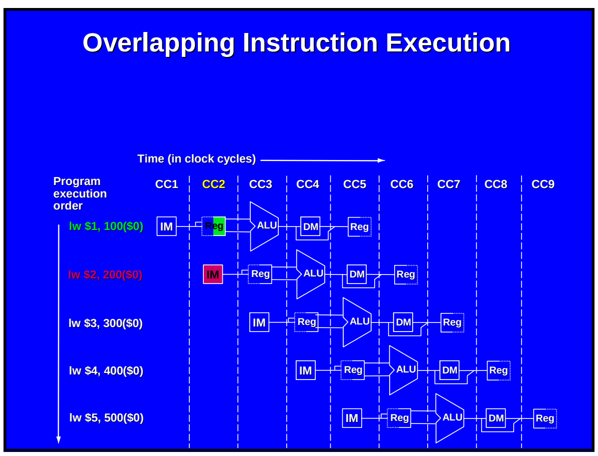

Execution: Branch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction Memory
Instruction
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Shift left 2
Add
ALU result
ALU
Zero
ALU result
16
Sign Extend
32
Address
Read data
Data Memory
Write data

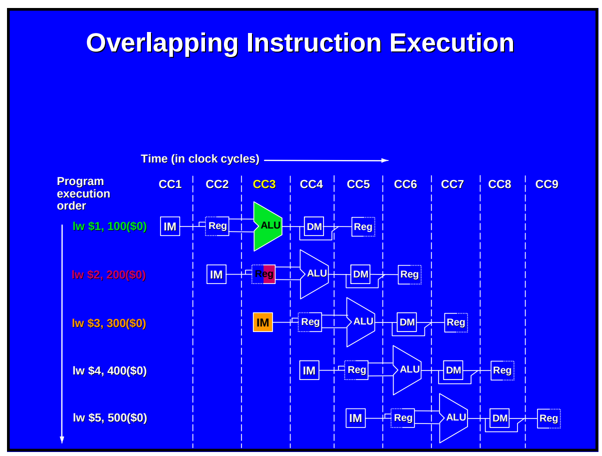

Memory: Branch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction Memory
Instruction
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Shift left 2
Add
ALU result
ALU
Zero
ALU result
16
Sign Extend
32
Address
Read data
Data Memory
Write data

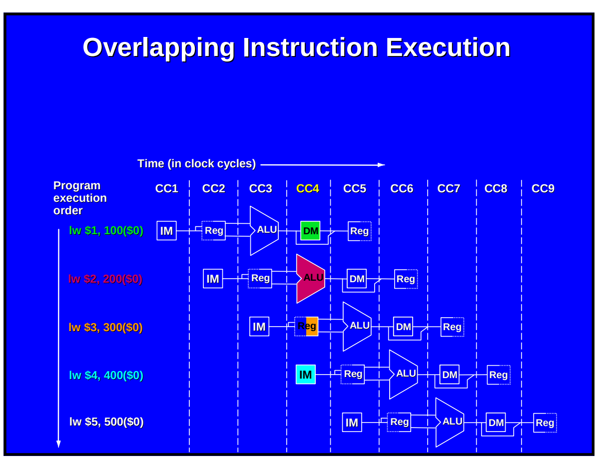

Write Back: Branch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction Memory
Instruction
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Shift left 2
Add
ALU result
ALU
Zero
ALU result
16
Sign Extend
32
Address
Read data
Data Memory
Write data

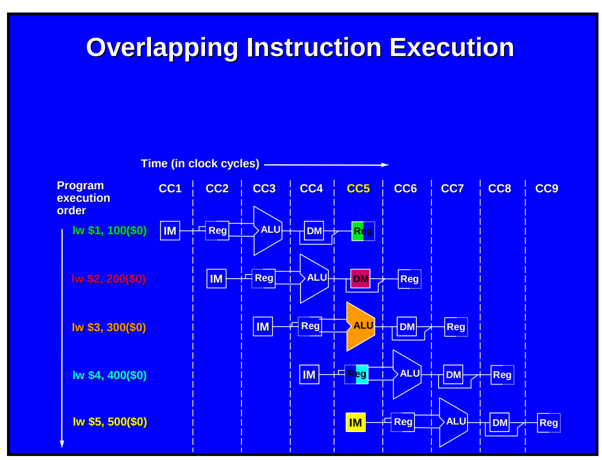

Overlapping Instruction Execution
Parts of different instruction execute at each pipeline stage during each cycle
Time (in clock cycles)
Program execution order
CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8 CC9
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
lw $4, 400($0)
lw $5, 500($0)
IM Reg ALU DM Reg


Overlapping Instruction Execution
Time (in clock cycles)
Program execution order
CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8 CC9
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
lw $4, 400($0)
lw $5, 500($0)
IM Reg ALU DM Reg

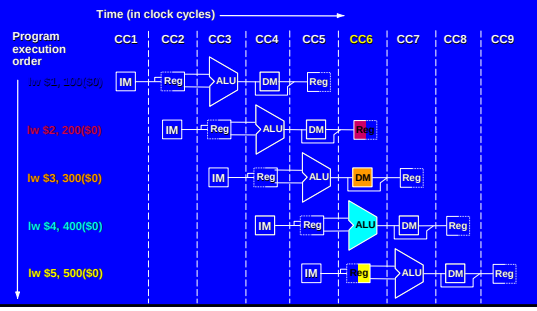

Overlapping Instruction Execution
Time (in clock cycles)
Program execution order
CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8 CC9
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
lw $4, 400($0)
lw $5, 500($0)
IM Reg ALU DM Reg


Overlapping Instruction Execution
Time (in clock cycles)
Program execution order
CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8 CC9
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
lw $4, 400($0)
lw $5, 500($0)
IM Reg ALU DM Reg

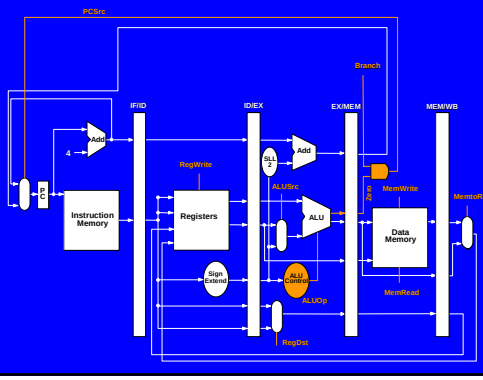

Overlapping Instruction Execution
Time (in clock cycles)
Program execution order
CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8 CC9
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
lw $4, 400($0)
lw $5, 500($0)
IM Reg ALU DM Reg


Overlapping Instruction Execution
Time (in clock cycles)
Program execution order
CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8 CC9
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
lw $4, 400($0)
lw $5, 500($0)
IM Reg ALU DM Reg

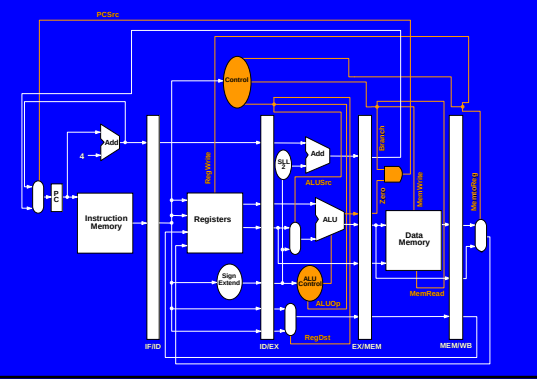
Overlapping Instruction Execution



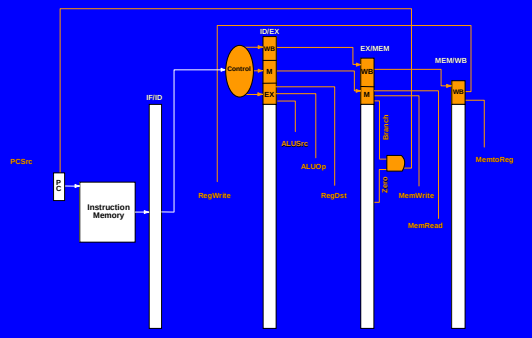
Control Lines



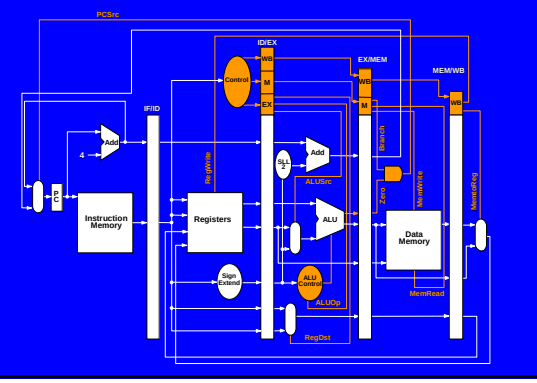
Control Unit



Pipelined Control



Pipeline with Pipelined Control

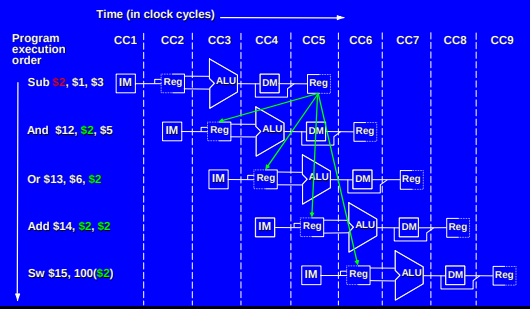


Data Hazards

- Data hazards can occur when:
  1. Instructions I_1 and I_2 are both in the pipeline
  2. I_2 follows I_1
  3. I_1 produces a result that is used by I_2
- The problem: because I_1 has not posted its result to the register file, I_2 may read an obsolete value.

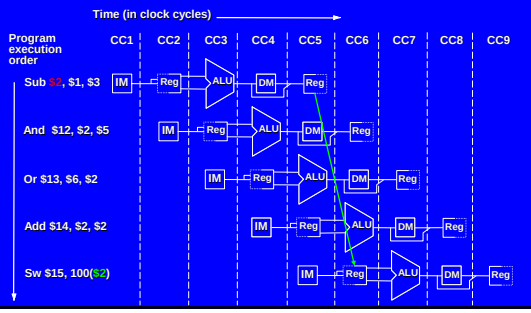
Data Hazards

$2 is not written by sub until CC5, but following instructions depend on $2 as soon as CC3



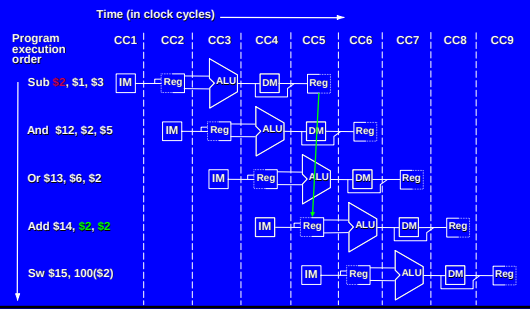
Data Hazards

Sw requires $2 at CC6, and $2 is written to register file at CC5, so not a problem



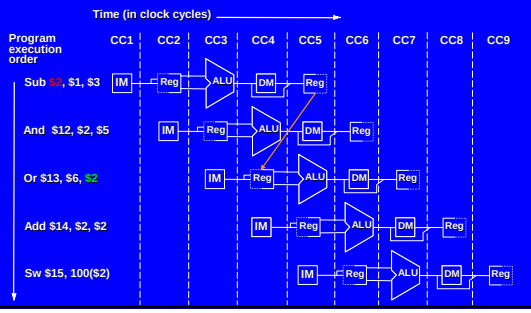
Data Hazards

Add requires $2 at CC5.5, and $2 is written to register file at CC5.0, so not a problem



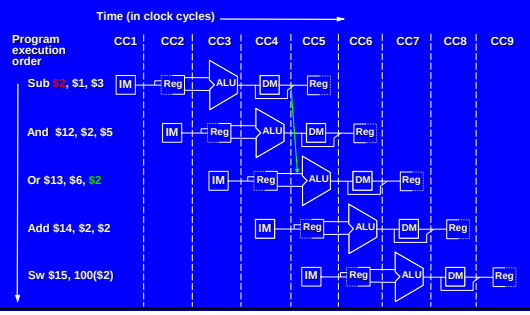
Data Hazards

Or requires $2 at CC4, and $2 is written to register file at CC5.0: hazard



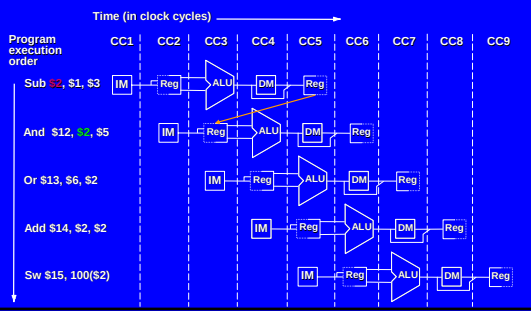
Data Hazards

However $2 is available at CC5 from internal pipeline latch



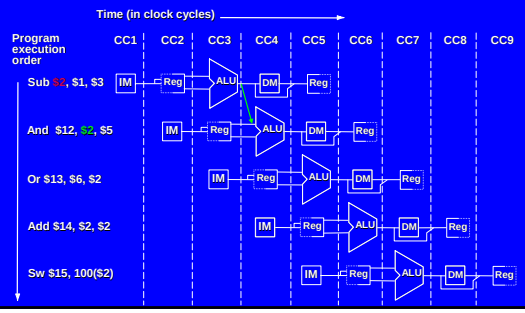
Data Hazards

And requires $2 at CC3, and $2 is written to register file at CC5.0: hazard



Data Hazards

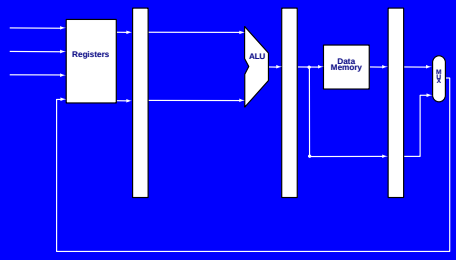
However $2 is available at CC4 from internal pipeline latch



Forwarding

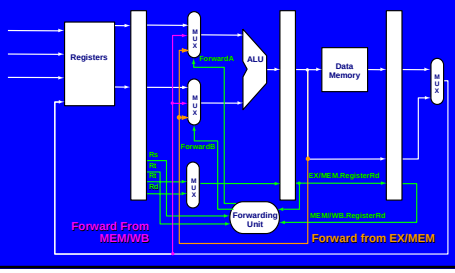
- Most data hazards can be resolved by forwarding, sending the internal pipeline result from I_1 to the stage where it is needed by I_2 which is following in the pipeline
- Result is still posted to the register file when I_1 reaches WB stage
- Forwarding requires hardware modifications to the datapath

DataPath without Forwarding



DataPath with Forwarding

- Forwarding data paths for results from Memory stage and WriteBack stage to Execution stage
- New ALU MUX to select operand from register file or newest result via forwarding



Fowarding Path Control

- Control used to decide whether to forward from the Memory stage

```
If  ((EX/MEM.RegWrite) #must be writing a register
and (EX/MEM.RegisterRd != 0) #$0 is always up to date
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)
#stage 4 result and stage 3 source operand match
  ForwardA = 10

If  ((EX/MEM.RegWrite)
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 10
```

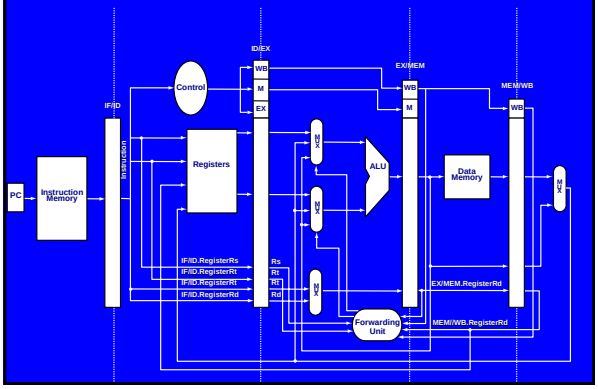
Fowarding Path Control

- Control used to decide whether to forward from the Writeback stage:

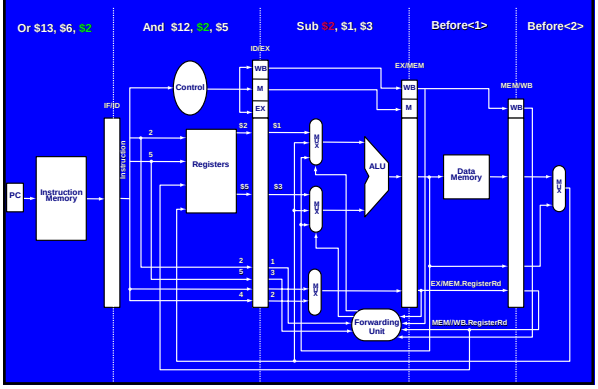
```
If  ((MEM/WB.RegWrite) #must be writing a register
and (MEM/WB.RegisterRd != 0) #$0 is always up to date
and !((EX/MEM.RegisterRd = ID/EX.RegisterRs) and
     (EX/MEM.RegWrite))
# forward result from stage 4 is newer, has priority
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
#stage 4 result and stage 3 source operand match
     ForwardA = 01

If  ((MEM/WB.RegWrite)
and (MEM/WB.RegisterRd != 0)
and !((EX/MEM.RegisterRd = ID/EX.RegisterRt) and
     (EX/MEM.RegWrite))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
     ForwardB = 01
```

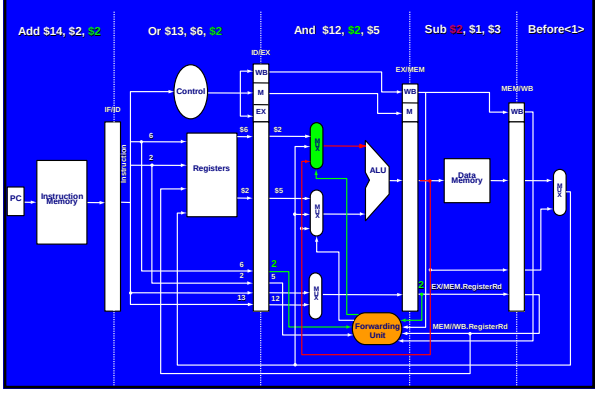
Complete Datapath with Forwarding



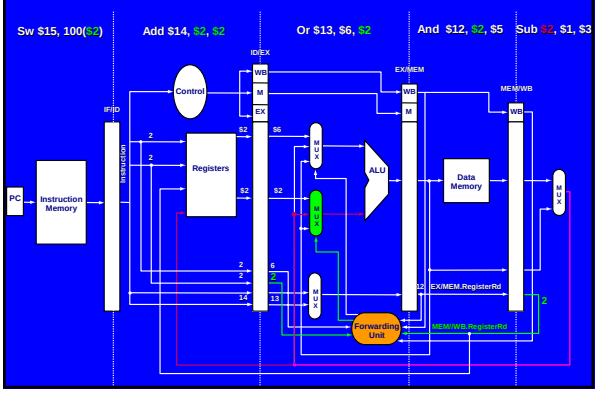
Forwarding (CC3)



Forwarding (CC4)

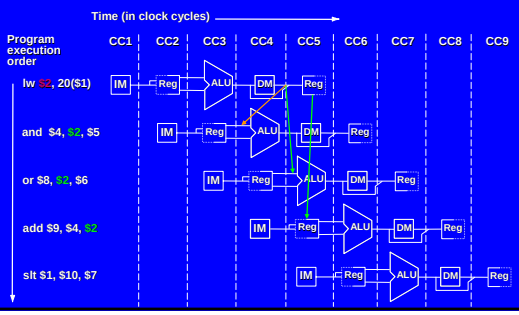


Forwarding (CC5)



Load Data Hazards

For loads most data hazards can be resolved with forwarding but some cannot



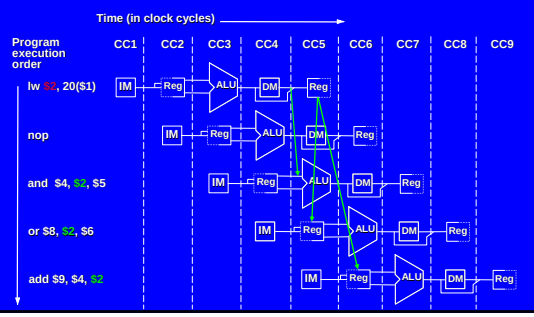
Load Hazard Solution #1

- Have compiler insert a NOP to space instructions further apart

```
lw  $2,20($1)
nop
and $4,$2,$5
or  $8,$2,$6
add $9,$4,$2
slt $1,$10,$7
```

Load Data Hazards

nop resolves load data hazard



Load Hazard Solution #2

- Best solution: compiler reorders (schedules) instructions to resolve hazard
  - SLT is independent of all instruction except LW, can move up to resolve hazard w/o nop

```
lw  $2,20($1)          lw  $2,20($1)
nop                    slt $1,$10,$7
and $4,$2,$5           and $4,$2,$5
or  $8,$2,$6           or  $8,$2,$6
add $9,$4,$2           add $9,$4,$2
slt $1,$10,$7
```

Load Hazard Solution #3

- Have the pipeline detect hazard and automatically inject a nop
  - reduces size of program
  - reduces cache misses
- LW and instruction before advance in next CC, following instructions stall, nop injected into EX

	IF	ID	EX	MEM	WB
CC_i:	or	and	lw	before 1	before 2
CC_{i+1}:	or	and	~~and~~	lw	before 1

Detecting a Load Hazard

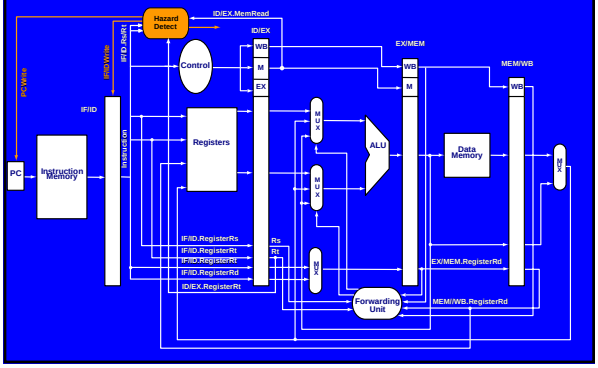
- Need to add an new hazard detection unit. Load hazard occurs under this condition:

```
If  (ID/EX.MemRead) #LW in stage 3
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
     or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
    #stage 2 Rs or Rt uses LW result
    stall the pipeline
```

- This logic is correct but not precise (creates too many stalls). Why?

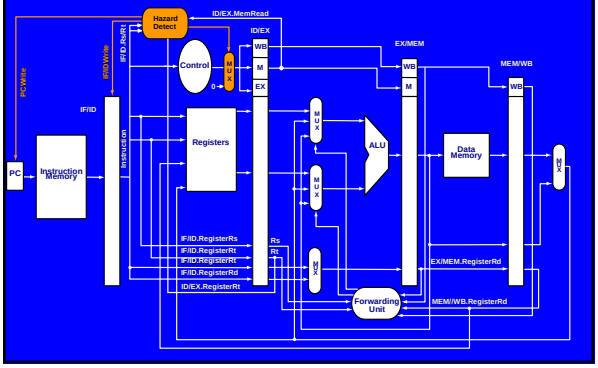
Hazard Detection Unit

Detection occurs in ID pipeline stage



Injecting A Nop (Pipeline Bubble)

Set instruction's control bits to all zeros; stall IF, ID

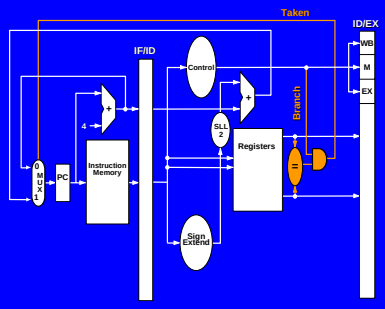


Refined NOP Injection

- Setting all control bits to zero is correct but is overkill. What is the minimum set of control bits that must be set to zero?

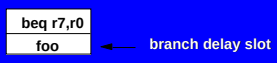
Early Branch Resolution

- Branch resolved in stage 2 to reduce penalty

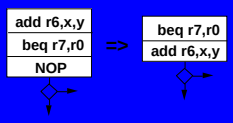


Delayed Branches

- Further reduce branch penalty by delaying the effect of branch one cycle I.e., always execute instruction after branch before PC = target

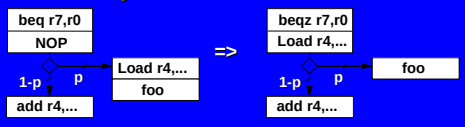


- Delay slot instruction must be independent of branch, "safe".
- Independent instruction can be moved into delay slot from (a) above the branch, (b) from taken path, or (c) from not-taken path

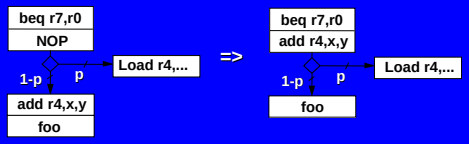


(a) from above: best because instruction always useful

Delayed Branches Cont.



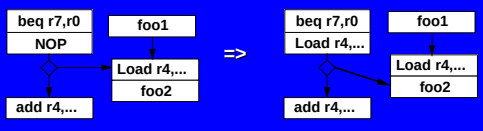
(b) from taken path: because lacking statistics for specific branch, taken is more likely (e.g., 2/3). Less valuable than (a) because fraction of useful work is p rather than 1



(c) from not-taken path: Useful work fraction is 1-p

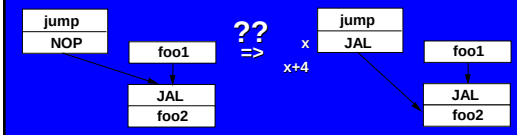
Branch Target Copying

- If branch target instruction is accessed via other program paths, Instruction must be copied, not moved.
- Branch retargeted to instruction after former target
- Dynamic instruction count decreases, although static does not



Branches in Delay Slot

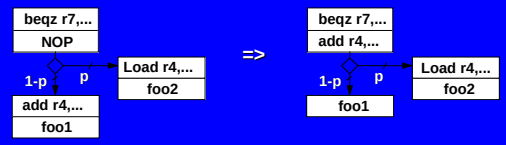
- Delay slot following Jump can almost always be filled by moving or copying target
- Branch generally cannot fill delay slot.



- Results in return to wrong address!!!

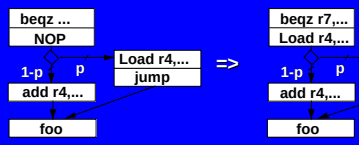
Profiled Delayed Branch

- Where safe instruction can be moved from taken or not-taken, fill delay slot based on profile statistics, maximizes useful work. E.g., for p = 0.38:

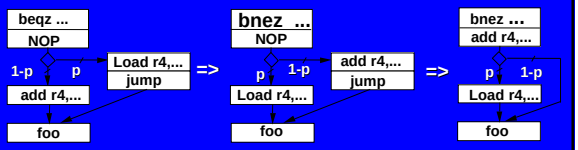


More Branch Optimizations

- Jump is eliminated if it is the only instruction left in block after move, => can eliminate two instructions. E.g., p=0.4

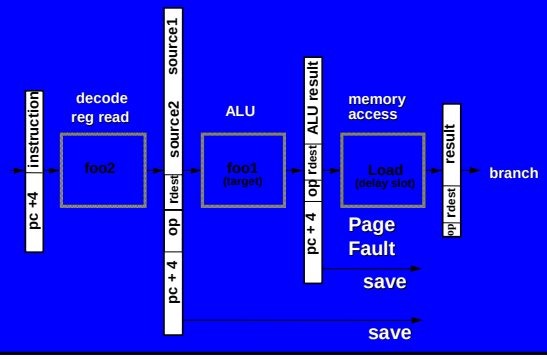


- Better still is to reverse the sense of the branch:



Delayed Branches & Exception

- Delayed branches must save PCs of next two instructions on exception, rather than one



Delayed Branching Performance

- For typical integer codes a useful instruction is executed in delay slot about 70% of the time thus branch cost is:

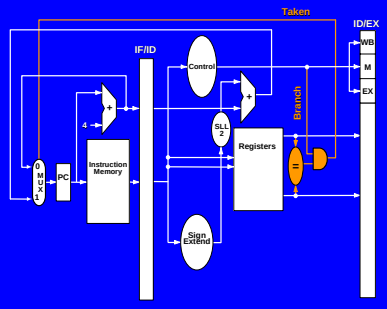
  (cycles lost per branch = 0.3) x
  (fraction of branches = 0.25) =
  8% performance loss

  Significant reduction from 75%, but still want smaller branch cost

  NOTE:
  if you consider slides 46&47 as the baseline, then CPI for branch instruction is 4 in baseline implementation and 75% performance loss is correct. This is a bit different than what we discussed in class.

Early Branch Resolution

- Branch resolved in stage 2 to reduce penalty



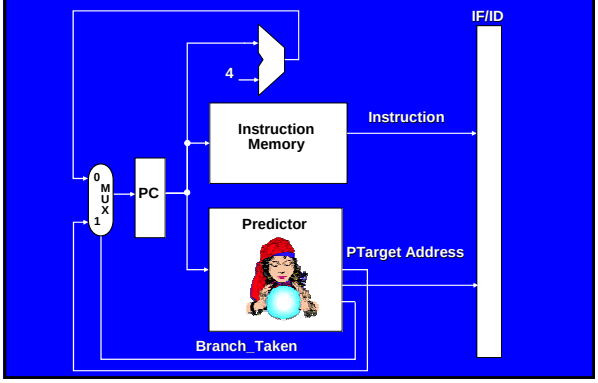
Branch Prediction

- Use a 'magic box' to:
  - Determine the instruction being fetched is a branch
  - Predict the branch outcome (taken/not taken)
  - Produce the target address

While the instruction is being fetched!

Branch Prediction

- Resolve branch in IF stage if prediction is correct



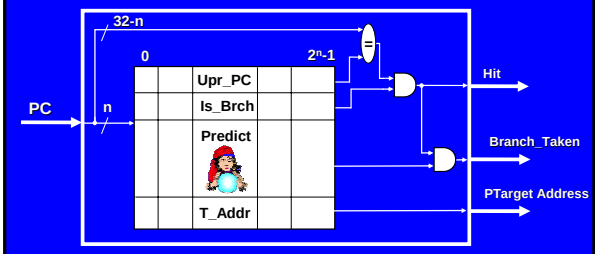
Simple Branch Prediction Table

Branch Target Buffer (BTB)

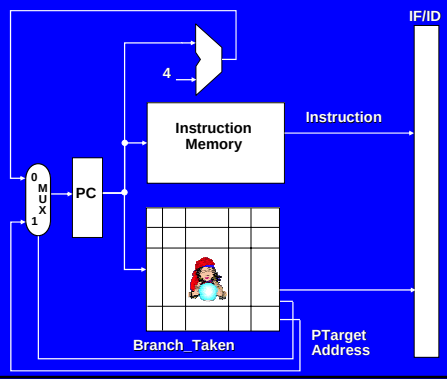
- Predictor module is a 2^n entry table indexed by n lower PC address bits. Fields are:
  1. 32-n upper PC address bits
  2. IsBranch bit
  3. Target Address (30 bits)
  4. Prediction
- Table was filled during past executions of the branch

Branch Target Buffer

- Because BTB is small, faster than instruction memory

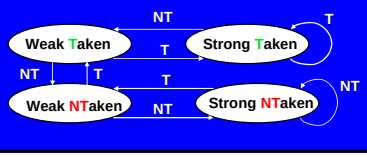


BTB in Context



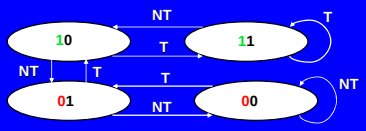
Two-Bit Branch Predictor

- Branch history using four states (two bits) avoids loop problem:
  - 00 strong not-taken
  - 01 weak not-taken
  - 10 weak taken
  - 11 strong taken
- Implemented as a state machine



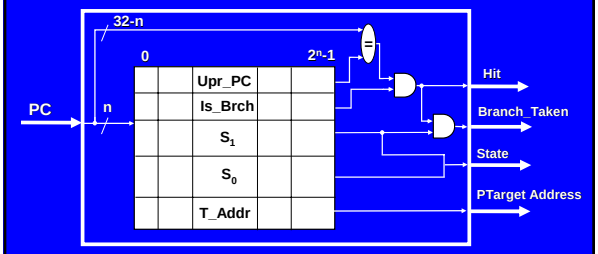
Two-Bit Branch Predictor

- Can think of this predictor as a saturating counter: +1 for taken, -1 for not taken

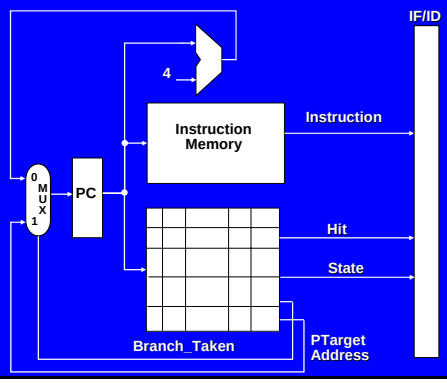


Branch Target Buffer Complete

- Because BTB is small, faster than instruction memory

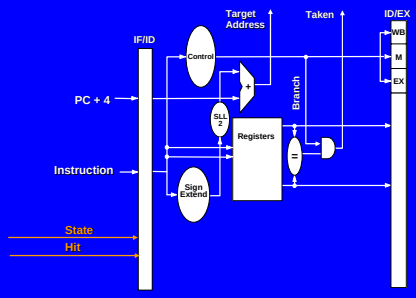


Final BTB in Context



Confirming Branch Prediction

- Prediction must be confirmed in ID stage, branch-history state updated

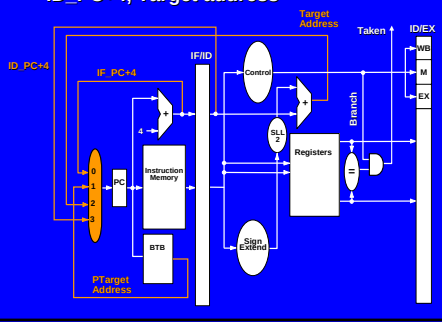


Confirming Branch Prediction

- Must consider pipeline action for following cases:
  1. No hit, no branch: do nothing
  2. No hit, branch:
     a. Nullify IF instruction
     b. Branch to computed target address
  3. Hit, condition = prediction: do nothing
  4. Hit, condition = taken, prediction = not taken:
     a. Nullify IF instruction
     b. Branch to computer target address
  5. Hit, condition = not taken, prediction = taken
     a. Nullify IF instruction
     b. Branch to ID stage PC+4

Next PC using Branch Prediction

- Using branch prediction now have four choices for next PC: IF_PC+4, PTarget address, ID_PC+4, Target address



BTB Updates

- Must consider BTB updates for following cases:
  1. No hit, no branch: do nothing
  2. No hit, branch:
     - Write new BTB entry, set prediction to weak condition
  3. Hit, condition = taken: ++prediction
  4. Hit, condition = not taken: --prediction