U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

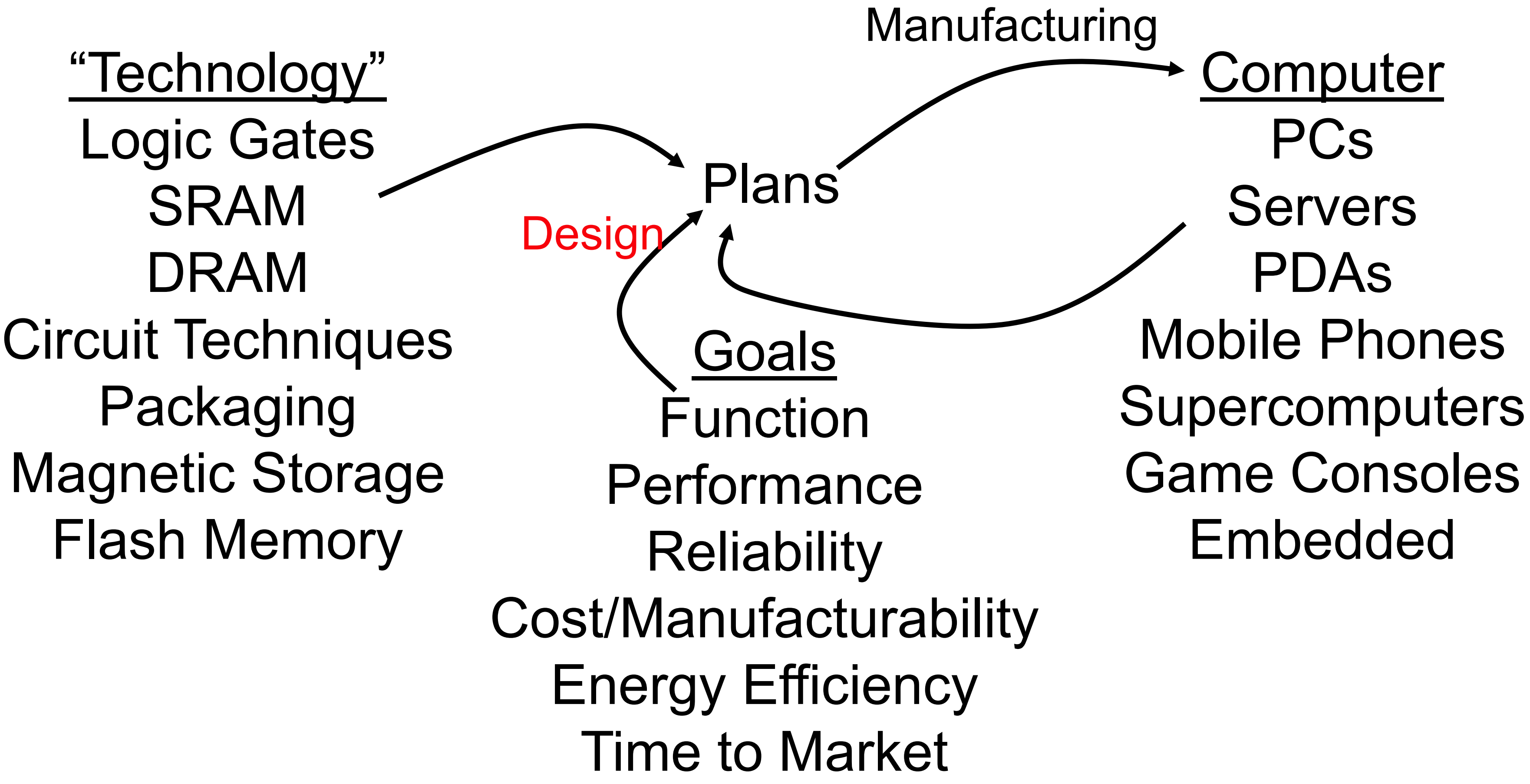
Prof. Karu Sankaralingam

Introduction (Chapter 1)

www.cs.wisc.edu/~karu/cs552/

Slides combined and enhanced by Mark D. Hill from work
by Falsafi, Marculescu, Nagle, Patterson, Roth, Rutenbar,
Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Role of Computer Architect



<u>"Technology"</u>

Logic Gates
SRAM
DRAM
Circuit Techniques
Packaging
Magnetic Storage
Flash Memory

Design

Plans

Manufacturing

<u>Goals</u>

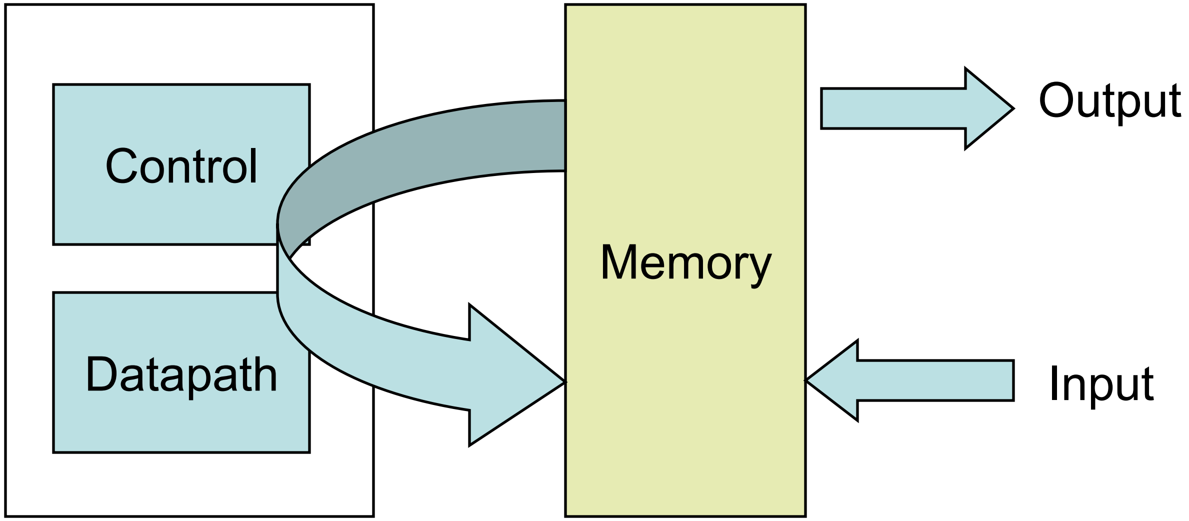
Function
Performance
Reliability
Cost/Manufacturability
Energy Efficiency
Time to Market

<u>Computer</u>

PCs
Servers
PDAs
Mobile Phones
Supercomputers
Game Consoles
Embedded

Basic Division of Hardware

- In space and time
  - In space



Basic Division of Hardware

- In time
  - Fetch the instruction from memory add r1, r2, r3
  - Decode the instruction - what does this mean?
  - Read input operands read r2, r3
  - Perform operation add
  - Write results write to r1
  - Determine next instruction pc := pc + 4

Moore's Law(s)

- Technologists will double # transistors per chip doubles every two years (or 18 months)
- Or architects will double performance per chip doubles every two years (or 18 months)
- These can't go on forever, but don't underestimate a trillion dollar industry

More Recent Microprocessor

- Intel Pentium4 [2003]
  - 32/64-bit data
  - 55M transistors
  - 0.90 μm CMOS
  - 3.4 GHz
  - 1.2 V
  - 101 mm^2

Building computer chips

- Complex multi-step process
  - slice ingots -> wafers
  - process wafers (many steps) -> patterned wafers
  - dice patterned wafers -> dies
  - test dies -> good dies
  - bond good die to package -> packaged dies (parts)
  - test parts -> good parts
  - ship to customers -> make money!

U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Instructions (Chapter 2)

www.cs.wisc.edu/~karu/cs552/

Slides combined and enhanced by Mark D. Hill from work
by Falsafi, Marculescu, Nagle, Patterson, Roth, Rutenbar,
Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Instruction Set Architecture (ISA)

- The **“contract”** between software and hardware
  - **Functional definition** of operations, modes, and storage locations supported by hardware
  - **Precise description** of how software can invoke and access them

- Strictly speaking, ISA is the architecture
  - Informally, architecture is also used to talk about the big picture of implementation
  - Better to call this **micro-architecture**

Aspects of ISAs

1. The Von Neumann model
   - Implicit structure of all modern ISAs
2. Format
   - Length and encoding
3. Operations
4. Operand model
   - Where are operands stored and how do address them?
5. Datatypes and operations
6. Control

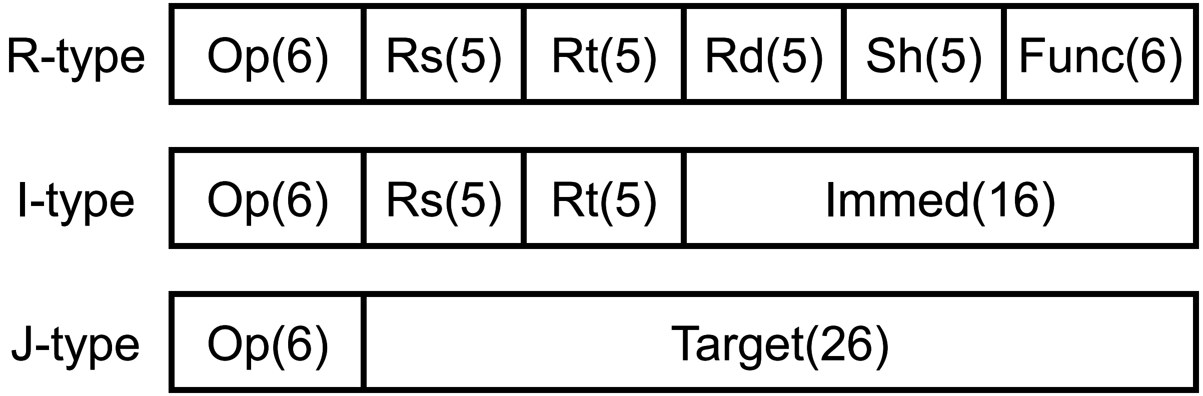
- Running example: MIPS
- Your project will use 16-bit MIPS-lite
- Touch on x86

(2) Instruction Format

- **Length**
  1. Fixed length
     - 32 or 64 bits (your project: 16 bit ISA)
     - + Simple implementation: compute next PC using only PC
     - – Code density
  2. Variable length
     - – Complex implementation
     - + Code density
  3. Compromise: two lengths
     - Example: MIPS$_{16}$
- **Encoding**
  - A few simple encodings simplify decoder implementation
  - Complex encoding can improve code density

MIPS Format

- Length
  - 32-bits
  - MIPS_{16}: 16-bit variants of common instructions for density
- Encoding
  - 3 formats, simple encoding
  - Q: how many operation types can be encoded in 6-bit opcode?



R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)
I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)		
J-type	Op(6)	Target(26)				

(4) Operations Act on Operands

- If you're going to add, you need at least 3 operands
    - Two source operands, one destination operand
- Question #1: Where can operands come from?
- Question #2: And how are they specified?

- Running example: A = B + C
    - Several options for answering both questions

- Discuss: Memory-Only & Registers
- Not Discuss: Stack & Accumulator

Memory Addressing

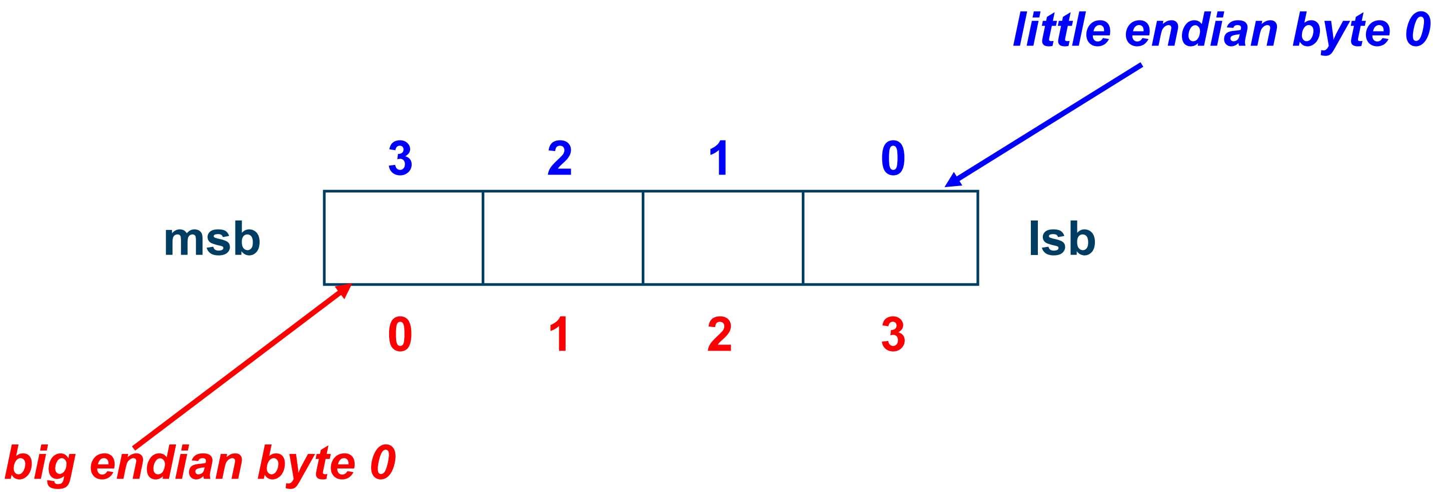
- ISAs assume **"virtual" address size**
  - Either 32 or 64 bits
  - Program can name 2^{32} bytes (4GB) or 2^{64} bytes (16PB)
  - ISA point? no room for even one address in a 32-bit instruction
- **Addressing mode**: way of specifying address
  - **Displacement: `ld R1,(R2)`** R1=mem[R2]
  - **Indirect: `ld R1,8(R2)`** R1=mem[R2+8]
  - **Index-base: `ld R1,(R2,R3)`** R1=mem[R2+R3]
  - **Memory-indirect: `ld R1,@(R2)`** R1=mem[mem[R2]]
  - **Auto-increment: `ld R1,(R2)+`** R1=mem[R2++]
  - **Scaled: `ld R1,(R2,R3,32,8)`** R1=mem[R2+R3*32+8]

  - What high-level program idioms are these used for?

Addressing Issue: Endian-ness

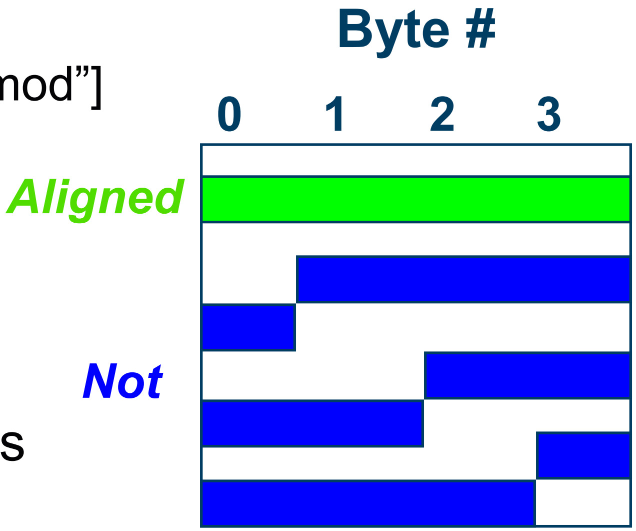
Byte Order

- Big Endian: byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, SPARC, HP PA-RISC
- Little Endian: byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC/Compaq Alpha



Another Addressing Issue: Alignment

- Alignment: require that objects fall on address that is multiple of their size
- 32-bit integer
  - Aligned if address % 4 = 0 [% is symbol for “mod”]
  - Aligned: `lw @XXXX00`
  - Not: `lw @XXXX10`
- 64-bit integer?
  - Aligned if ?
- Question: what to do with unaligned accesses (uncommon case)?
  - Support in hardware? Makes all accesses slow
  - Trap to software routine? Possibility
  - **MIPS? ISA support**: unaligned access using two instructions:
    `lw @XXXX10 = lwl @XXXX10; lwr @XXXX10`



(6) Control Instructions I

- One issue: **testing for conditions**
  - Option I: **compare and branch instructions**

    ```
    blti $1,10,target
    ```

    + Simple, – two ALUs: one for condition, one for target address
  - Option II: **implicit condition codes**

    ```
    subi $2,$1,10   // sets "negative" CC
    bn target
    ```

    + Condition codes set "for free", – implicit dependence is tricky
  - Option III: **condition registers, separate branch insns**

    ```
    slti $2,$1,10
    bnez $2,target
    ```

    – Additional instructions, + one ALU per, + explicit dependence

Control Instructions II

- Another issue: **computing targets**
  - Option I: **PC-relative**
    - Position-independent within procedure
    - Used for branches and jumps within a procedure
  - Option II: **Absolute**
    - Position independent outside procedure
    - Used for procedure calls
  - Option III: **Indirect** (target found in register)
    - Needed for jumping to dynamic targets
    - Used for returns, dynamic procedure calls, switches

  - How far do you need to jump?
    - Typically not so far within a procedure (they don’t get that big)
    - Further from one procedure to another

MIPS Control Instructions

- MIPS uses all three
  - PC-relative → conditional branches: **`bne`**, **`beq`**, **`blez`**, etc.
    - 16-bit relative offset, <0.1% branches need more
    - PC = PC + 4 + immediate if condition is true (else PC=PC+4)

I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)

  - Absolute → unconditional jumps: **`j target`**
    - 26-bit offset (can address 2^{28} words < 2^{32} → what gives?)

J-type	Op(6)	Target(26)

  - Indirect → Indirect jumps: **`jr $rd`**

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)

Control Instructions III

- Another issue: support for procedure calls?
  - We “link” (remember) address of the calling instruction + 4 (current PC + 4) so we can return to it after the procedure

- MIPS
  - Implicit return address register is **`$ra(=$31)`**
  - Direct jump-and-link: **`jal address`**

    → $ra = PC+4; PC = address
  - Can then return from call with: **`jr $ra`**

  - Or can call with indirect jump-and-link: **`jalr $rd, $rs`**

    → $rd = PC+4; PC = $rs // explicit return address register
  - Then return with: **`jr $rd`**

RISC vs. CISC

- **RISC**: reduced-instruction set computer
  - Coined by P+H in early 80’s
- **CISC**: complex-instruction set computer
  - Not coined by anyone, term didn’t exist before “RISC”

- Religious war (one of several) started in mid 1980’s
  - RISC (MIPS, Alpha) “won” the technology battles
  - CISC (IA32 = x86) “won” the commercial war
    - Compatibility a stronger force than anyone (but Intel) thought
    - Intel beat RISC at its own game … more on this soon

Intel x86: The Penultimate CISC (VAX ultimate)

  - Variable length instructions: 1-16 bytes
  - Few registers: 8 and each one has a special purpose
  - Multiple register sizes: 8,16,32 bit (for backward compatibility)
  - Accumulators for integer instrs, and stack for FP instrs
  - Multiple addressing modes: indirect, scaled, displacement
  - Register-register, memory-register, and memory-register insns
  - Condition codes
  - Instructions for memory stack management (push, pop)
  - Instructions for manipulating strings (entire loop in one instruction)

- Summary: yuck!

U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Arithmetic Part A (3.1-3.5, B.5-B.6)

www.cs.wisc.edu/~karu/cs552/

Slides combined and enhanced by Mark D. Hill from work by Falsafi, Marculescu, Nagle, Patterson, Roth, Rutenbar, Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Integer Representation

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Balance, number of zeros, ease of arithmetic

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - `1010   -> 0101 + 1 = 0110`
  - `0110   -> 1001 + 1 = 1010`

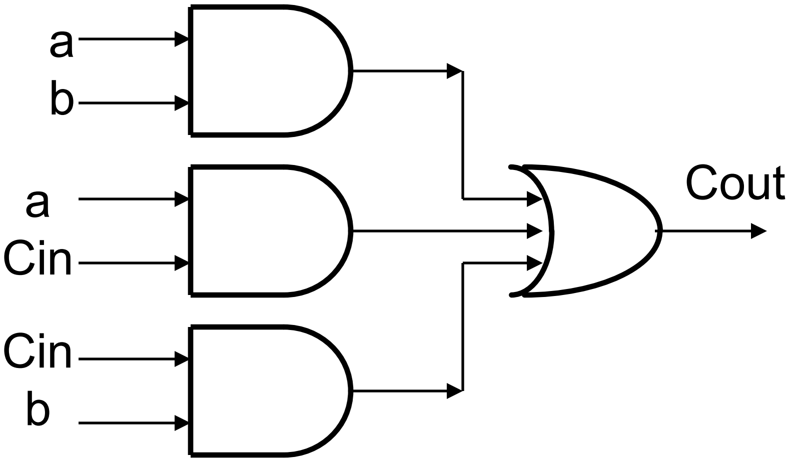
- Converting n bit numbers into numbers with more than n bits:
  - copy the most significant bit (the sign bit)

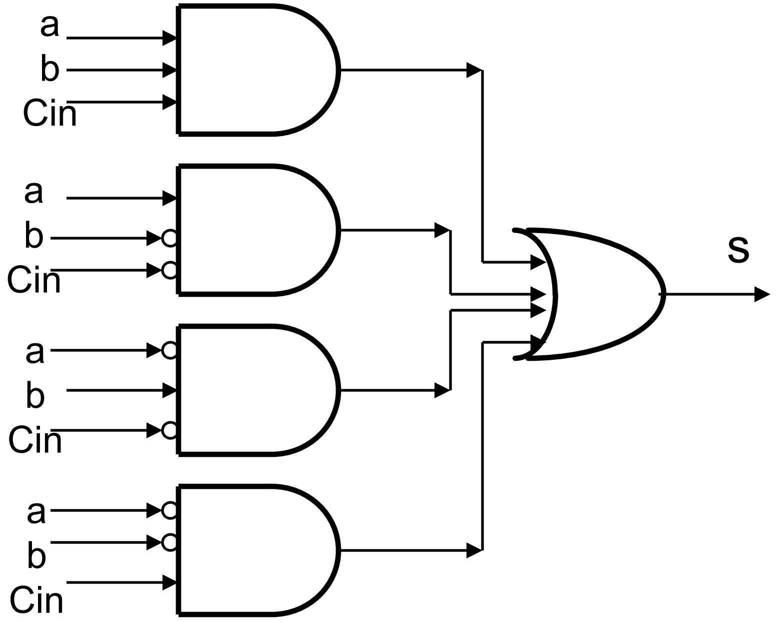
```
0010   -> 0000 0010
1010   -> 1111 1010
```

  - Called "sign extension"

Full adder

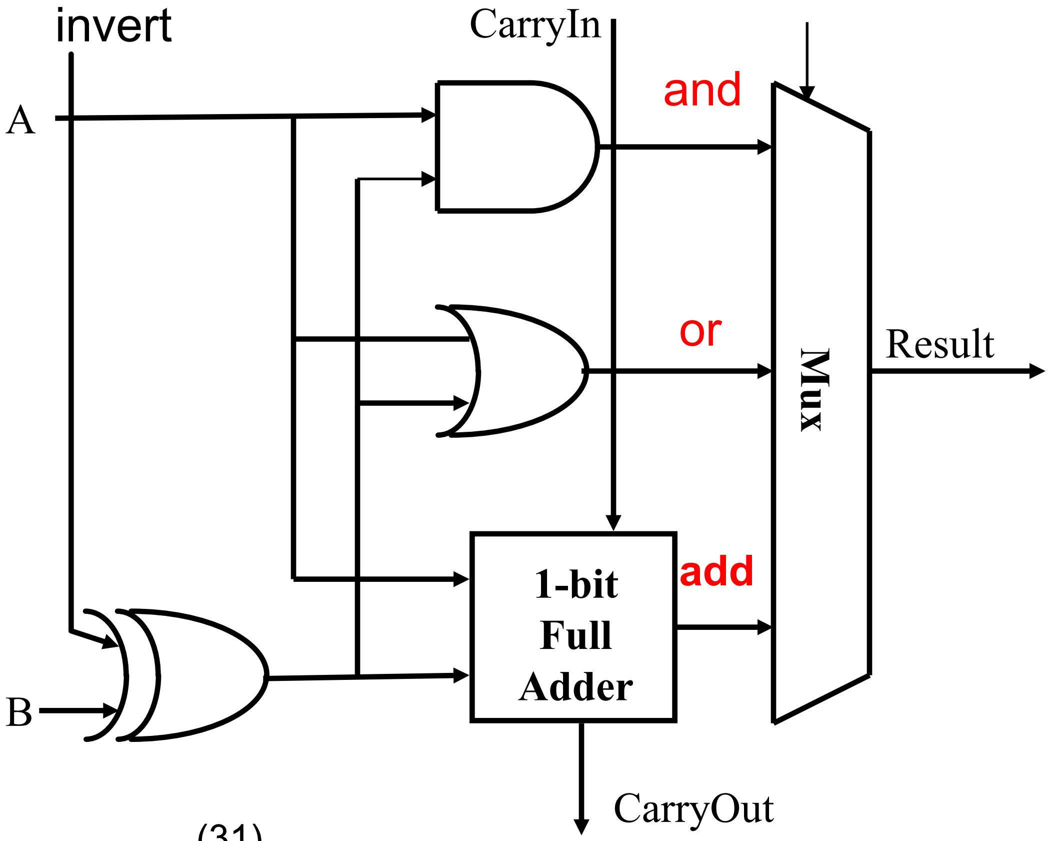
- Three inputs and two outputs
- Cout, s = F(a,b,Cin)
  - Cout : only if at least two inputs are set
  - S : only if exactly one input or all three inputs are set
- Logic?



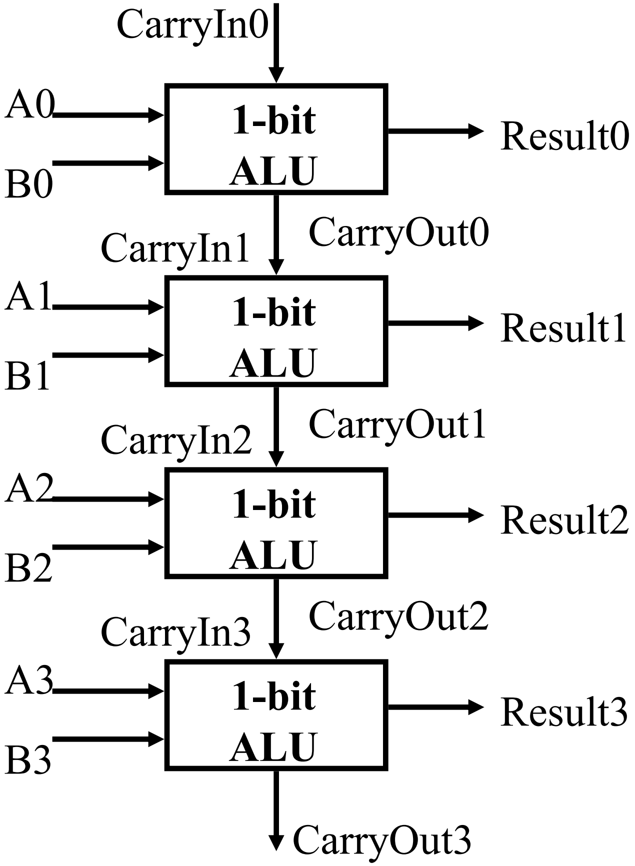


Subtract

- A - B = A + (– B)
  - form two complement by invert and add one



Ripple-carry adder



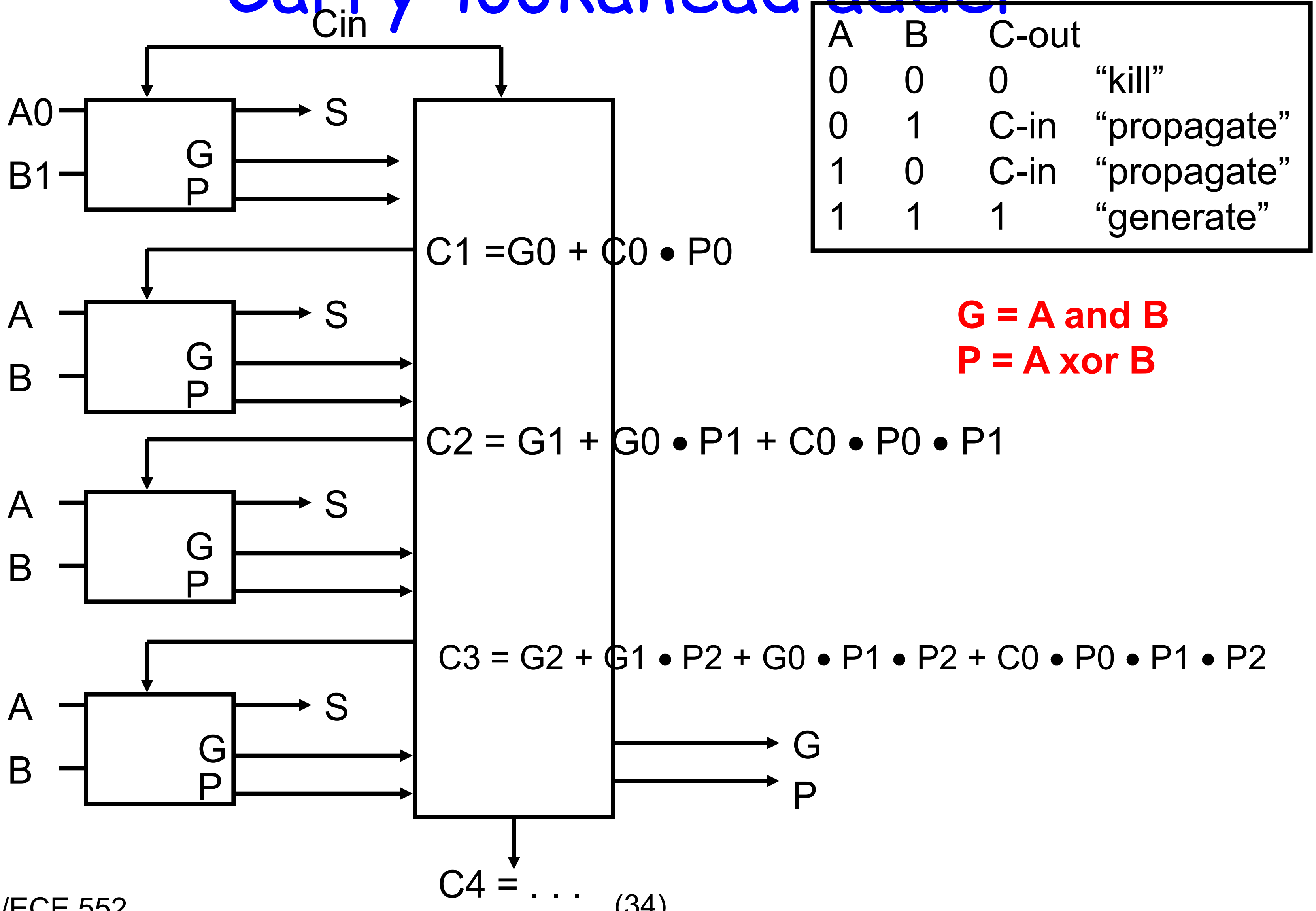
Carry look-ahead

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry?
    - $g_i = a_i b_i$
  - When would we propagate the carry?
    - $p_i = a_i + b_i$
- Did we get rid of the ripple?

Carry-lookahead adder

A	B	C-out	
0	0	0	“kill”
0	1	C-in	“propagate”
1	0	C-in	“propagate”
1	1	1	“generate”

G = A and B
P = A xor B



$C1 = G0 + C0 \bullet P0$

$C2 = G1 + G0 \bullet P1 + C0 \bullet P0 \bullet P1$

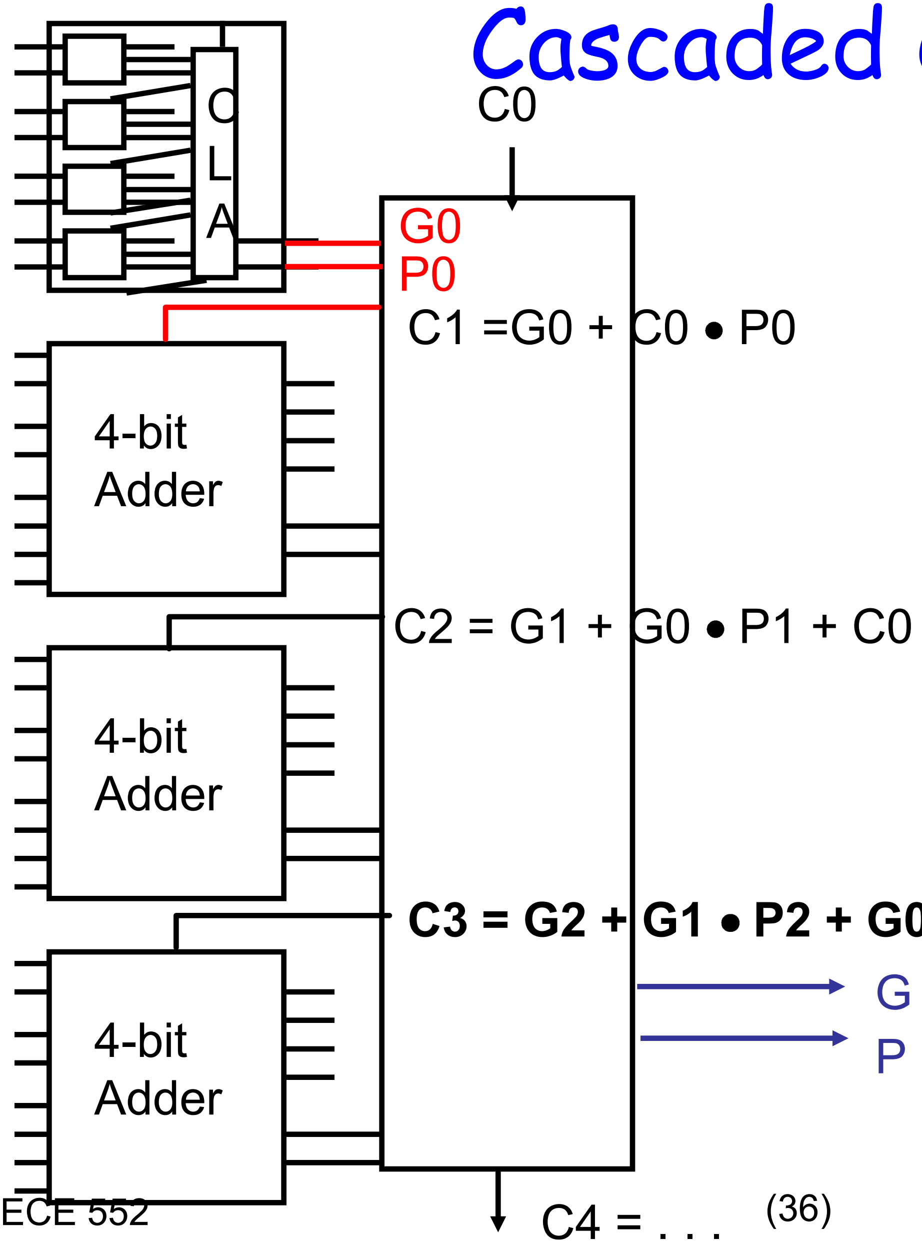
$C3 = G2 + G1 \bullet P2 + G0 \bullet P1 \bullet P2 + C0 \bullet P0 \bullet P1 \bullet P2$

C4 = . . .

Carry-Lookahead Adder

- Waitaminute!
  - Nothing has changed
  - Fanin problems if you flatten!
    - Linear fanin, not exponential
  - Ripple problem if you don't!
- Enables divide-and-conquer
- Figure out Generate and Propagate for 4-bits <u>together</u>
- <u>Compute hierarchically</u>

Cascaded CLA



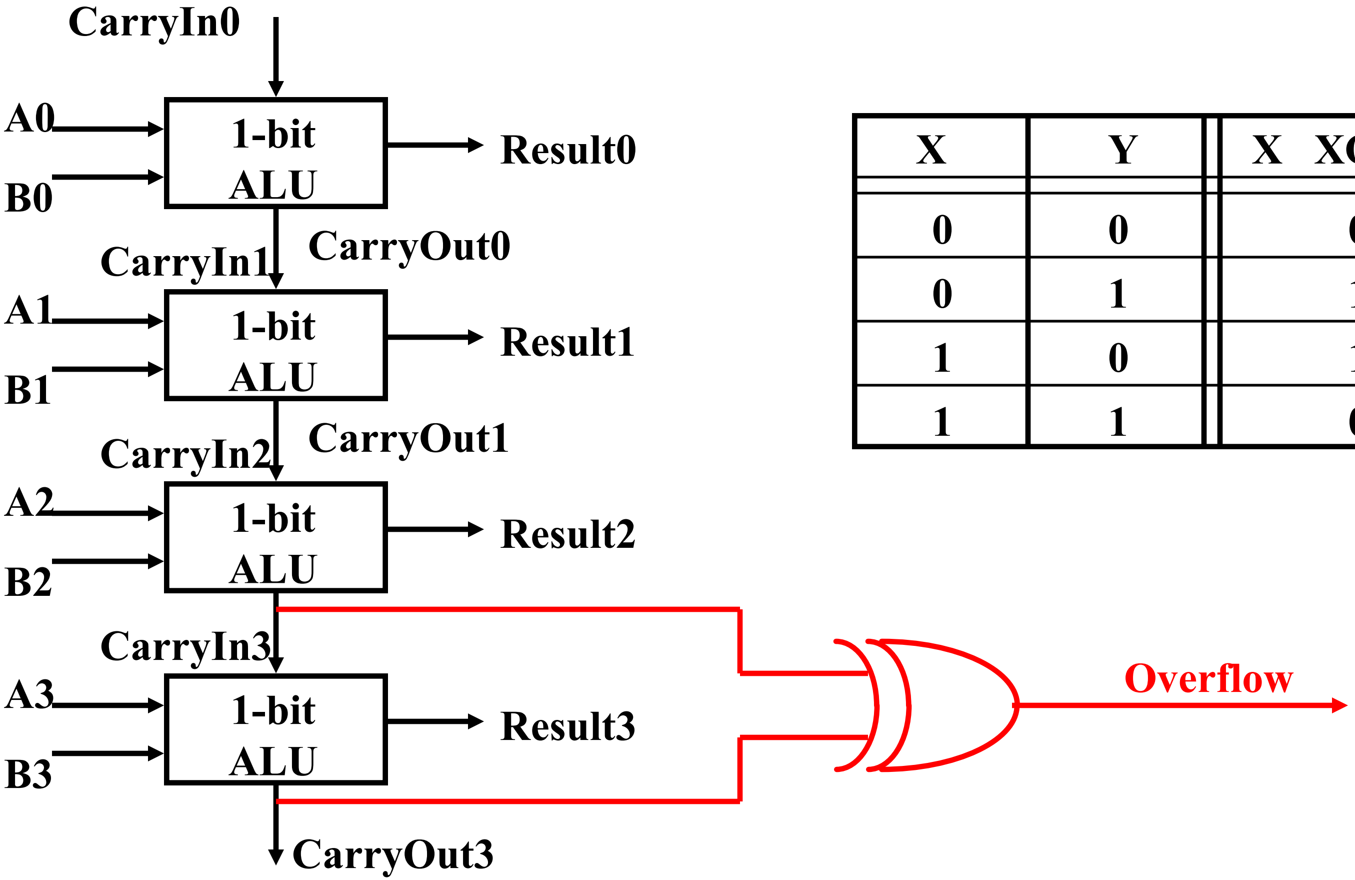
$C1 = G0 + C0 \bullet P0$

$C2 = G1 + G0 \bullet P1 + C0 \bullet P0 \bullet P1$

$\mathbf{C3 = G2 + G1 \bullet P2 + G0 \bullet P1 \bullet P2 + C0 \bullet P0 \bullet P1 \bullet P2}$

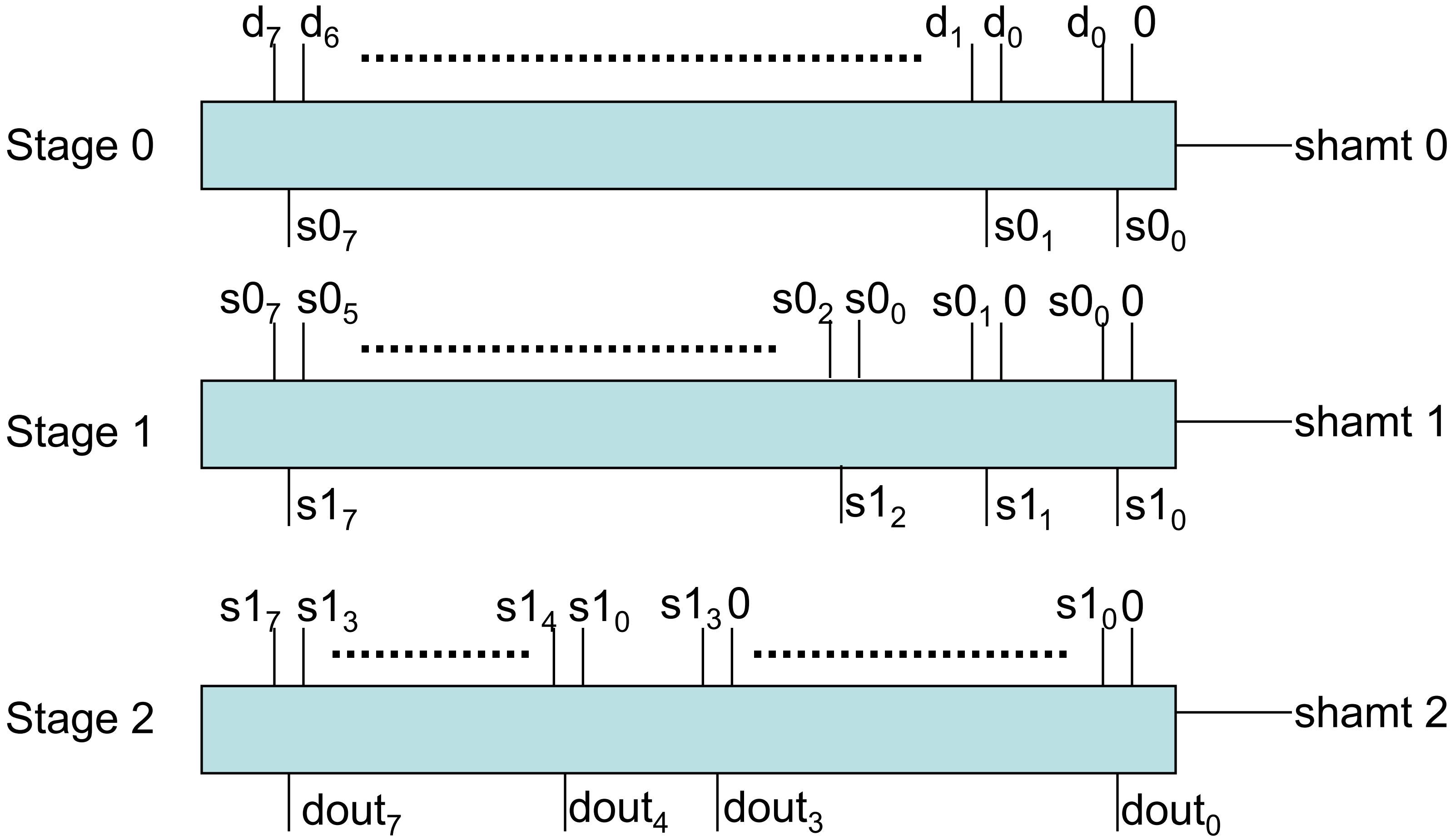
Overflow detection

- Carry into MSB ⊕ Carry out of MSB
  - For N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Barrel Shifter


d7 d6 .... d1 d0 d0 0
Stage 0
shamt 0
s07 s01 s00
s07 s05 .... s02 s00 s01 0 s00 0
Stage 1
shamt 1
s17 s12 s11 s10
s17 s13 .... s14 s10 s13 0 .... s10 0
Stage 2
shamt 2
dout7 dout4 dout3 dout0

U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Performance (Chapter 4)

www.cs.wisc.edu/~karu/cs552/

Slides combined and enhanced by Mark D. Hill from work by Falsafi, Marculescu, Nagle, Patterson, Roth, Rutenbar, Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Performance of Computers

- Want
  - Highest Performance (modeling oil fields)
  - Lowest Cost (doorknob)
  - Lowest Cost/Performance (most common)

- Performance will depend on workload
- Computers not completely interchangable
  - PC cannot (currently) have 128 GB memory

Defining Performance

- What is important to who?

1. Computer system user
   - minimize elapsed time for program = time_end - time_start
   - called response time
2. Computer center manager
   - maximize completion rate = #jobs/second
   - called throughput

Performance Comparison

- Machine A is n times faster than machine B iff
  - perf(A)/perf(B) = time(B)/time(A) = n

- Machine A is x% faster than machine B iff
  - perf(A)/perf(B) = time(B)/time(A) = 1 + x/100

- E.g., A 10s, B 15s
  - 15/10 = 1.5 => A is 1.5 times faster than B
  - 15/10 = 1 + 50/100 => A is 50% faster than B

Iron law

- Time/program =
  instrs/program x cycles/instr x sec/cycle
- sec/cycle (a.k.a. cycle time, clock time) - 'heartbeat' of computer
  - mostly determined by technology and CPU organization
- cycles/instr (a.k.a. CPI)
  - mostly determined by ISA and CPU organization
  - overlap among instructions makes this smaller
- instr/program (a.k.a. instruction count)
  - instrs executed NOT static code
  - mostly determined by program, compiler, ISA

Beware of Millions of Instr / Sec

- MIPS = instruction count/(execution time x 10^6)
  = clock rate/(CPI x 10^6) (How?)
- Often ignores program & quotes "peak"
  - ideal conditions => guarantee not to exceed!!
- Ignores instruction/program changes
  - E.g., adding floating-point H/W can hurt MIPS
  - 50 simple instructions replace by one slow FP op
- Okay if
  - instrs/program constant (e.g. same executable)
  - real program; not peak

Beware of Millions of FP Ops / Sec

- MFLOPS =
  FP ops in program/(execution time x 10^6)
- Assumes FP ops independent of compiler/ISA
  - Assumption not true
  - may not have divide instruction in ISA
  - optimizing compilers can remove
- Relative MIPS and normalized MFLOPS
  - adds to confusion! (see book)

Which Programs?

- Execution time of what?
- Best case - you always run the same set of programs
  - port them and time the whole “workload”
- In reality, use benchmarks
  - programs chosen to measure performance
  - predict performance of actual workload (hopefully)
  - saves effort and money
  - representative? honest?
  - Example Suites: EEMBC, MediaBench, SPEC, &TPC

How to Average

- Another: arithmetic mean (same result: B 9.1 times faster than A)
- Arithmetic mean of times: $\left\{ \sum_{=} \quad ne_i \right\} \quad n$ for n programs
- AM(A) = 1001/2 = 500.5
- AM(B) = 110/2 = 55
- 500.5/55 = 9.1

- Valid only if programs run equally often, else use "weight" factors
- Weighted arithmetic mean:

$$\left\{ \sum_{=} \quad ight_i \times \quad ime_i \right\} \quad n$$

Harmonic Mean

- Harmonic mean of rates = $$\frac{1}{\left\{\sum_{i=1} \frac{1}{rate_i}\right\} \Big/ n}$$
  - Use HM if forced to start and end with rates
- Trick to do arithmetic mean of times but using rates and not times

Geometric Mean

- Don’t use arithmetic mean on ratios (normalized numbers)
- Use geometric mean for ratios
  - geometric mean of ratios =
  - Use GM if forced to use ratios $\sqrt[n]{\prod_{i=1} \quad tio_i}$
- Independent of reference machine (math property)
- In the example, GM for machine A is 1, for machine B is also 1
- Normalized with respect to either machine
- Used in SPECint and SPECfp

Summary for Averages

- Use AM for times
- Use HM if forced to use rates
- Use GM if forced to use ratios

- Better yet
  - Use unnormalized numbers to compute time

Amdahl's Law

- Why does the common case matter the most?
- Let an optimization speed f fraction of time by a factor of s
- assuming that old time = T, what is the speedup?
  - f is the "affected" fraction of T
  - (1-f) is the unaffected fraction

- Speedup =
- =

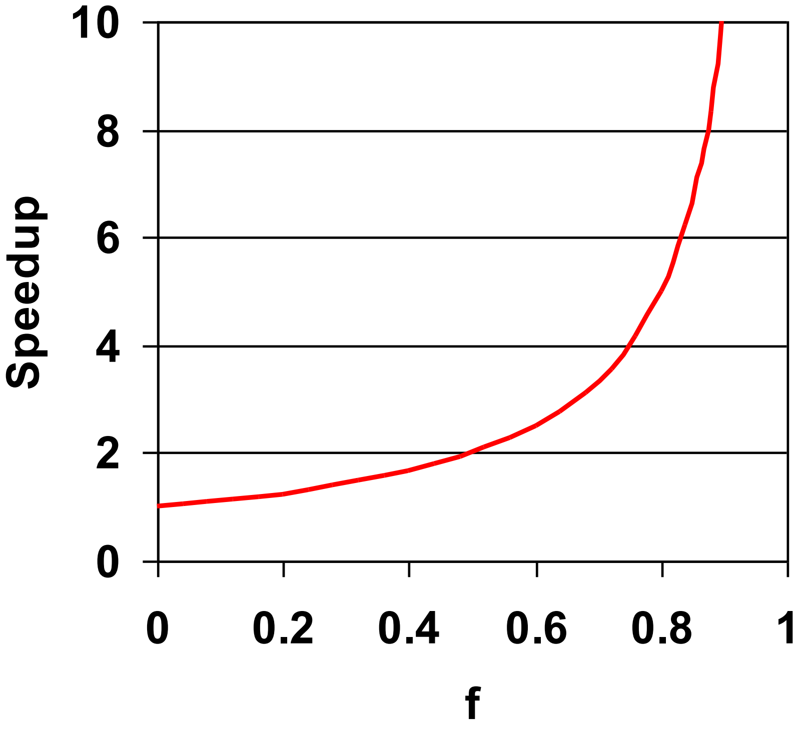
$$\frac{time_{old}}{time_{new}} = \frac{unaffected_{old} + affected_{old}}{unaffected_{new} + affected_{new}}$$

$$\frac{(1-f)\times T + f\times T}{(1-f)\times T + \frac{f}{s}\times T}$$

Amdahl's Law: Limit

- Make common case fast because:

$$\lim_{s\to}\left(\frac{1}{1-f+f/s}\right)=\frac{1}{1-f}$$



U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

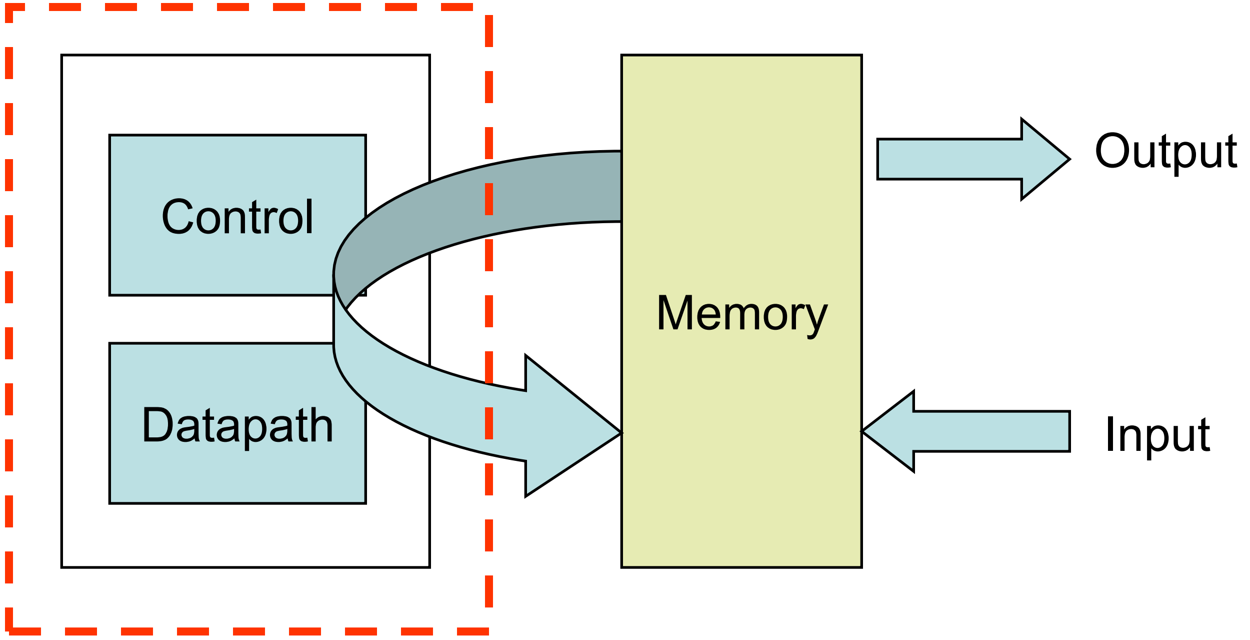
Prof. Karu Sankaralingam

Single-Cycle Processor (5.1-5.4)

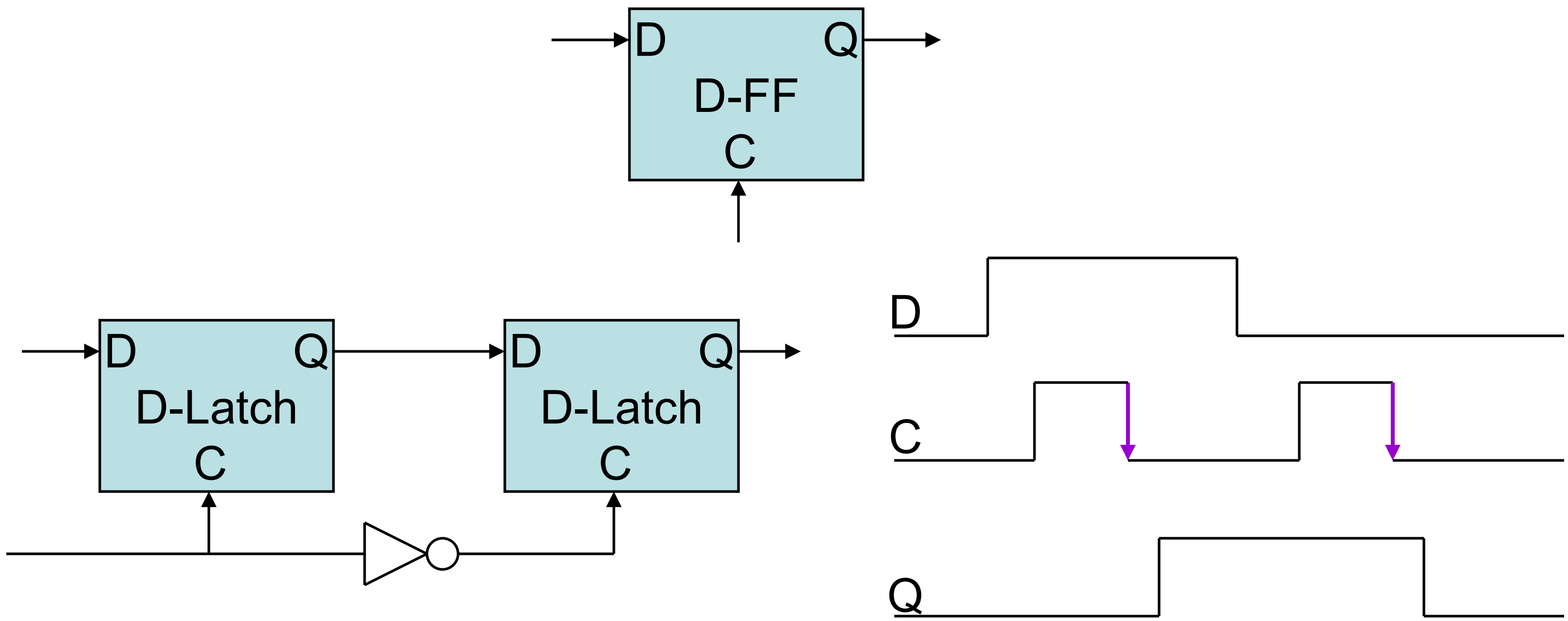
www.cs.wisc.edu/~karu/cs552/

Slides combined and enhanced by Mark D. Hill from work
by Falsafi, Marculescu, Nagle, Patterson, Roth, Rutenbar,
Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Processor Implementation

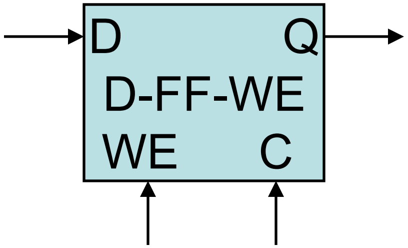


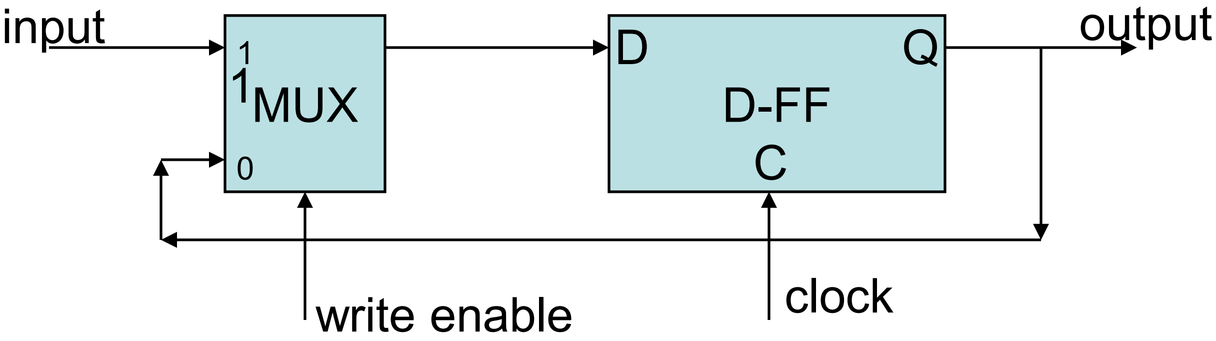
Review: D Flip-flop



- D flip-flop - built from 2 D-latches
  - while clock high, D flows into 1st latch, but not 2nd
  - in 2nd Q retains old value
- Remember D at falling edge & propagate thru 2nd latch

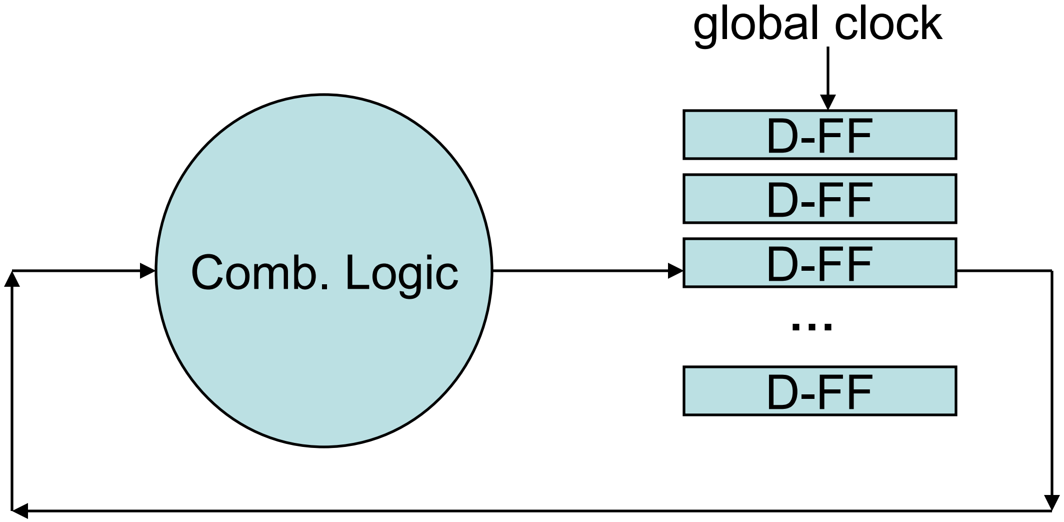
D-FF WriteEnable (preferred design)





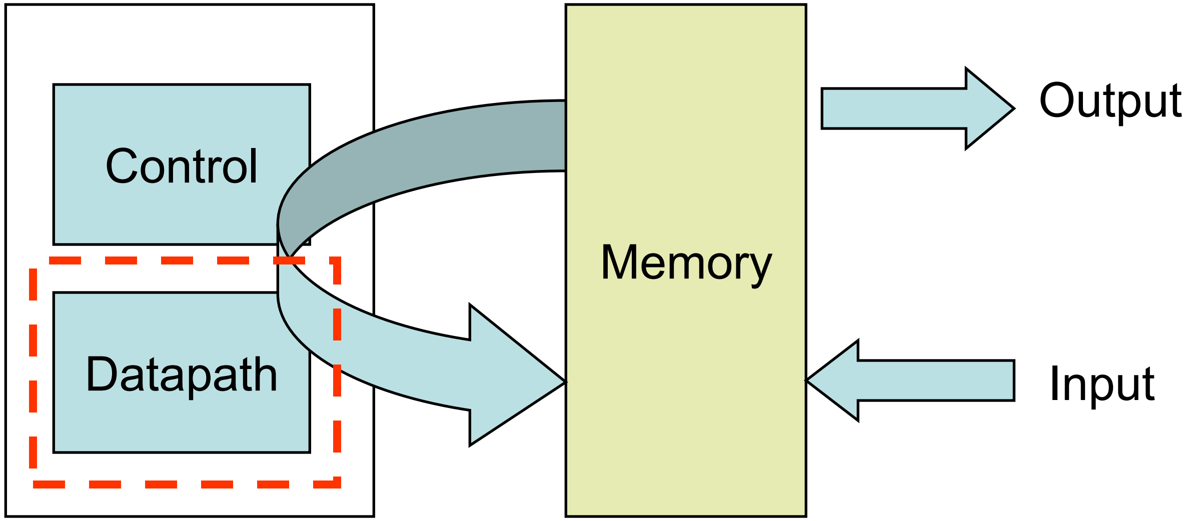
552 Clocking Methodology Rules

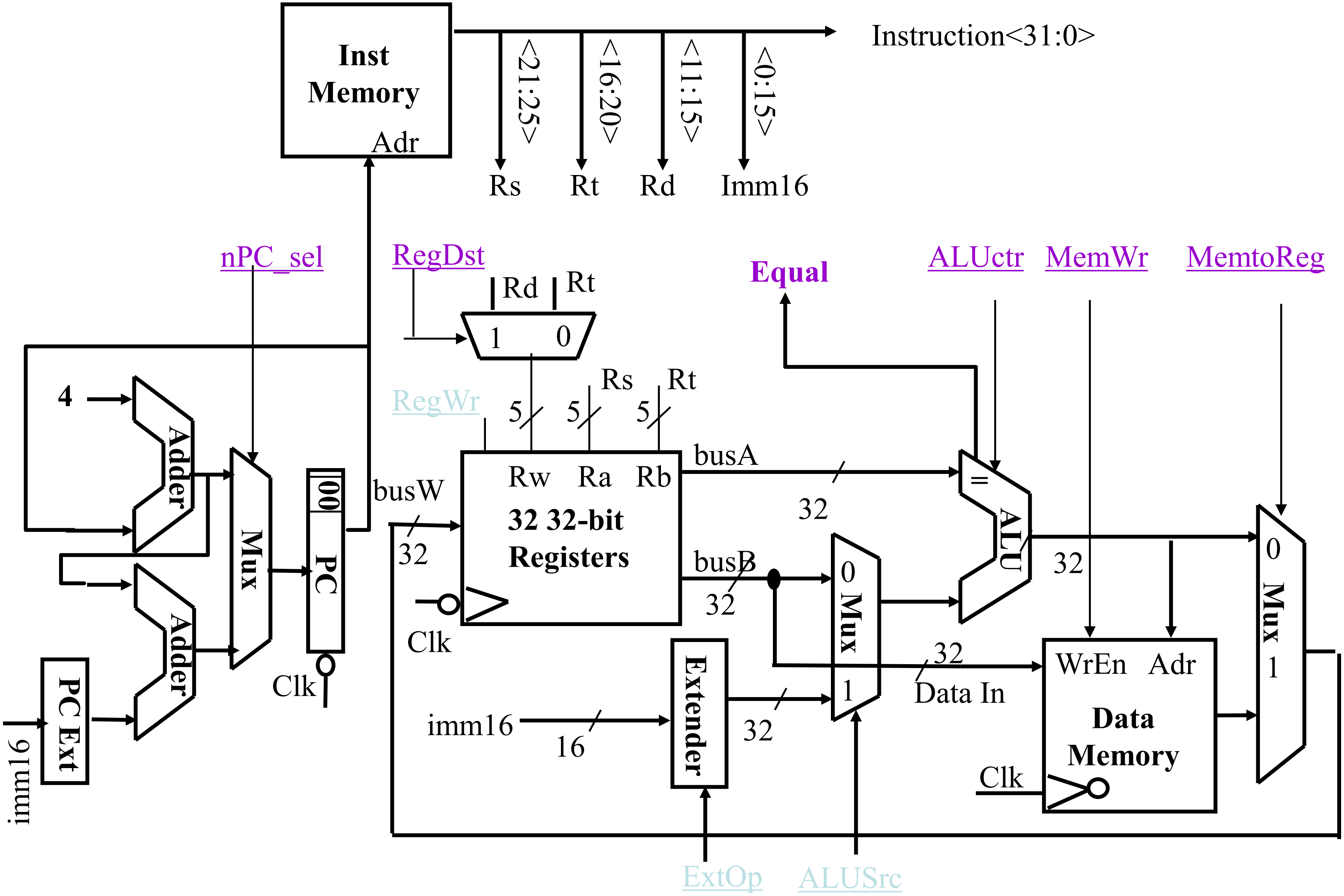
- We provide D-FF design
- Use this D-FF for all processor state
- Same unqualified clock for all D-FFs
- Combinational logic must finish in one cycle



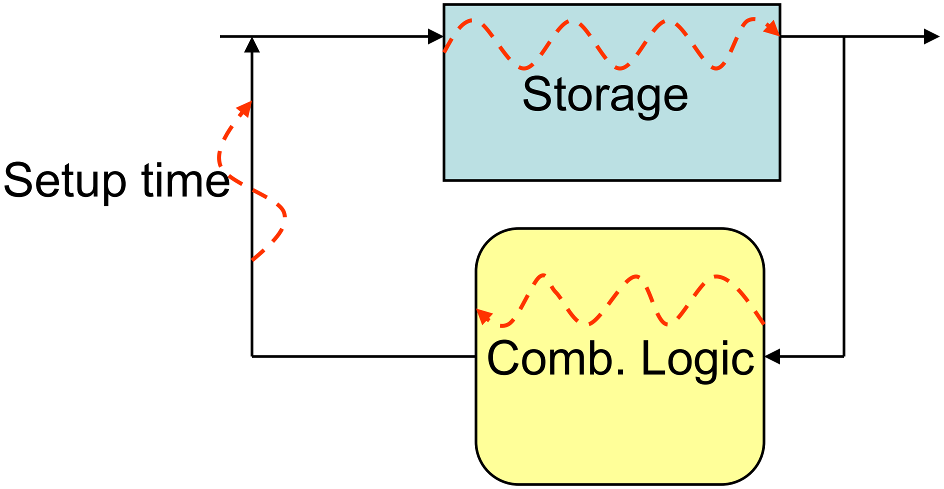
Processor Implementation

- Next : Single-Cycle Datapath




Inst
Memory
Adr
Instruction<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rs
Rt
Rd
Imm16
nPC_sel
RegDst
Rd
Rt
1
0
Equal
ALUctr
MemWr
MemtoReg
4
Adder
Mux
00
PC
Clk
Adder
PC Ext
imm16
RegWr
5
5
5
Rs
Rt
Rw
Ra
Rb
busW
32
32 32-bit
Registers
Clk
busA
32
busB
32
=
ALU
0
Mux
1
32
32
Data In
WrEn
Adr
Data
Memory
Clk
0
Mux
1
32
imm16
16
Extender
32
ExtOp
ALUSrc

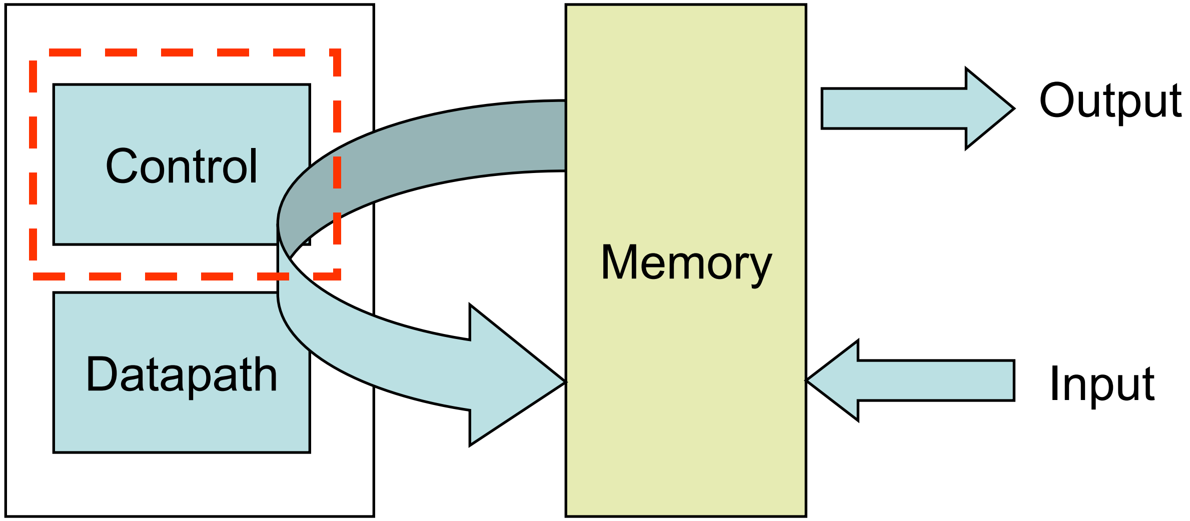
Cycletime



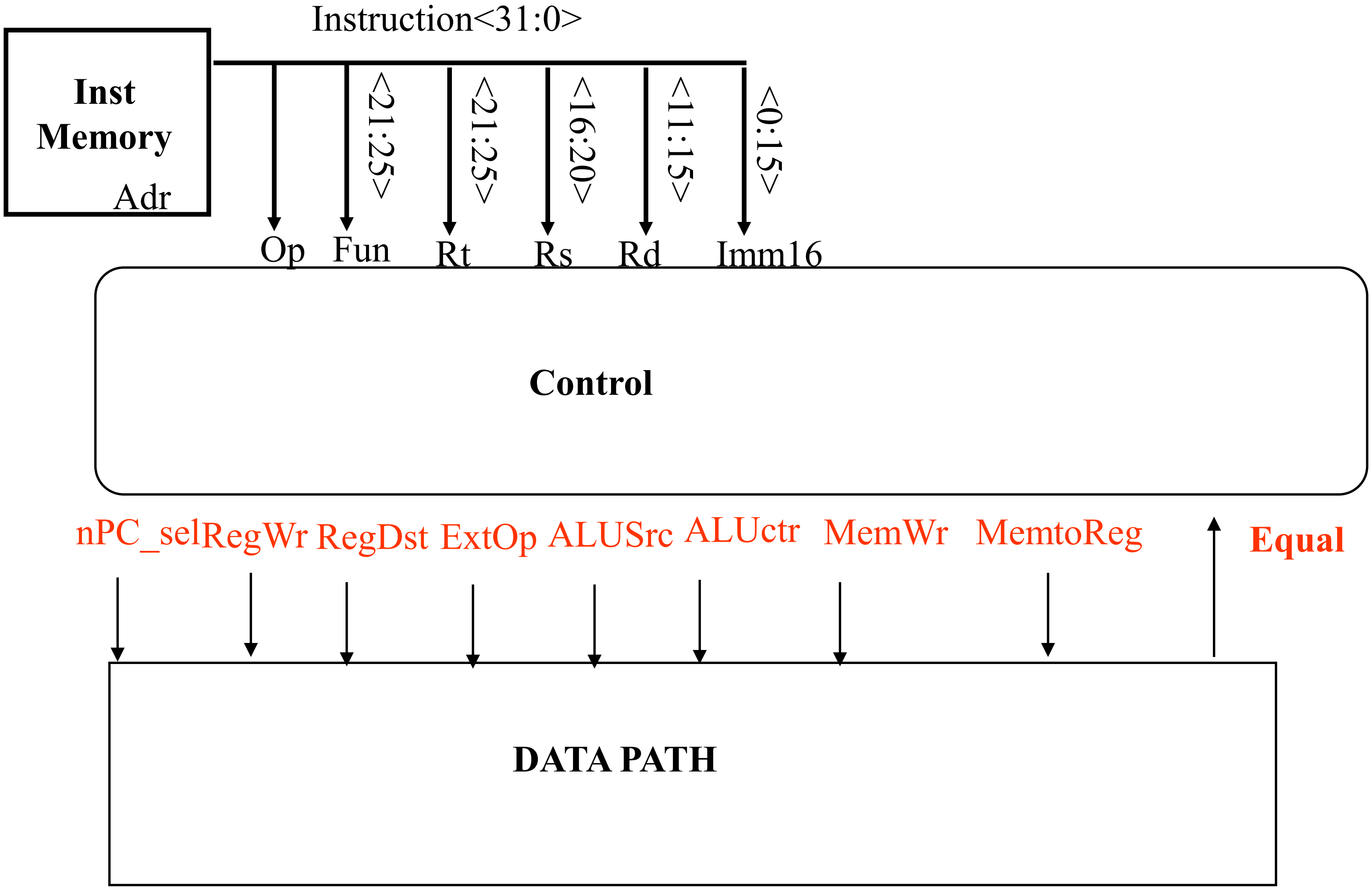
- What should the clock period be?
  - Enough to compute the next state values
    - Propagation clk-to-Q (new state)
    - Comb. Logic delay
    - Setup requirements

Processor Implementation

- Next : Control for Single-Cycle Datapath

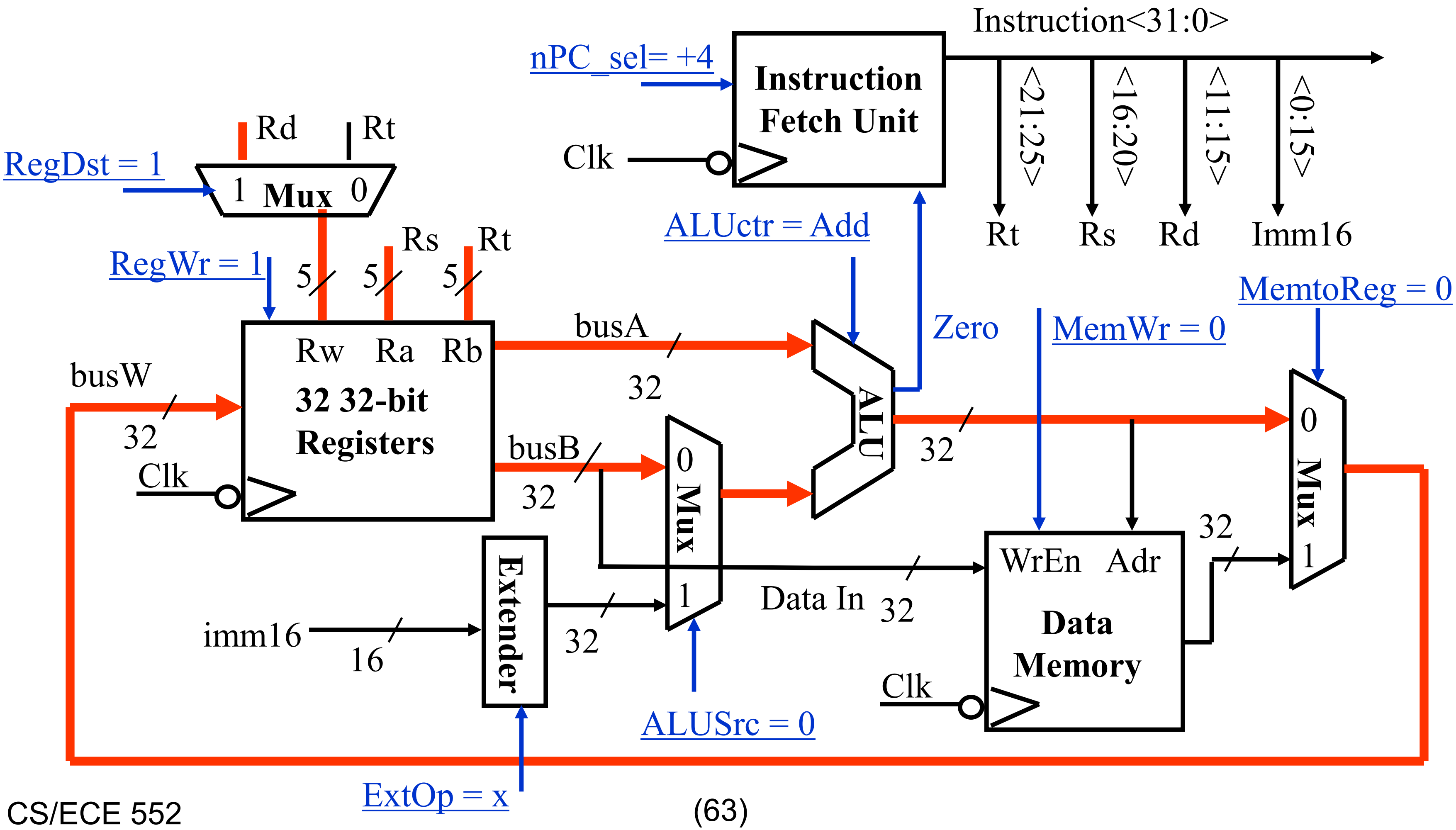


Control for Datapath



Controls for Add Operation

- R[rd] = R[rs] + R[rt]

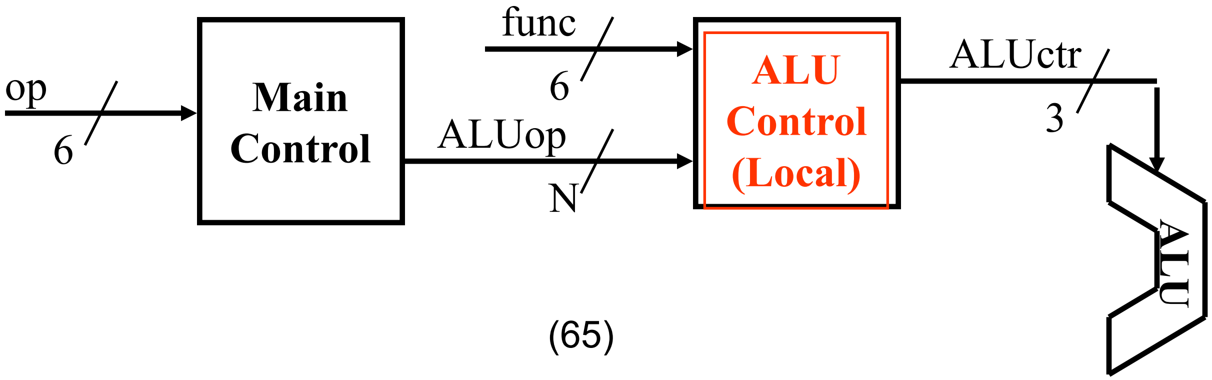


Controls: Logic equations

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == "R-type") then "regB"
  elseif (OP == BEQ) then regB, else "imm"
- ALUctr <= if (OP == "R-type") then **funct**
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub" else
  "add"
- ExtOp <= if (OP == ORi) then "zero" else "sign"
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

Global Control: Truth Table

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



Truth Table for RegWrite

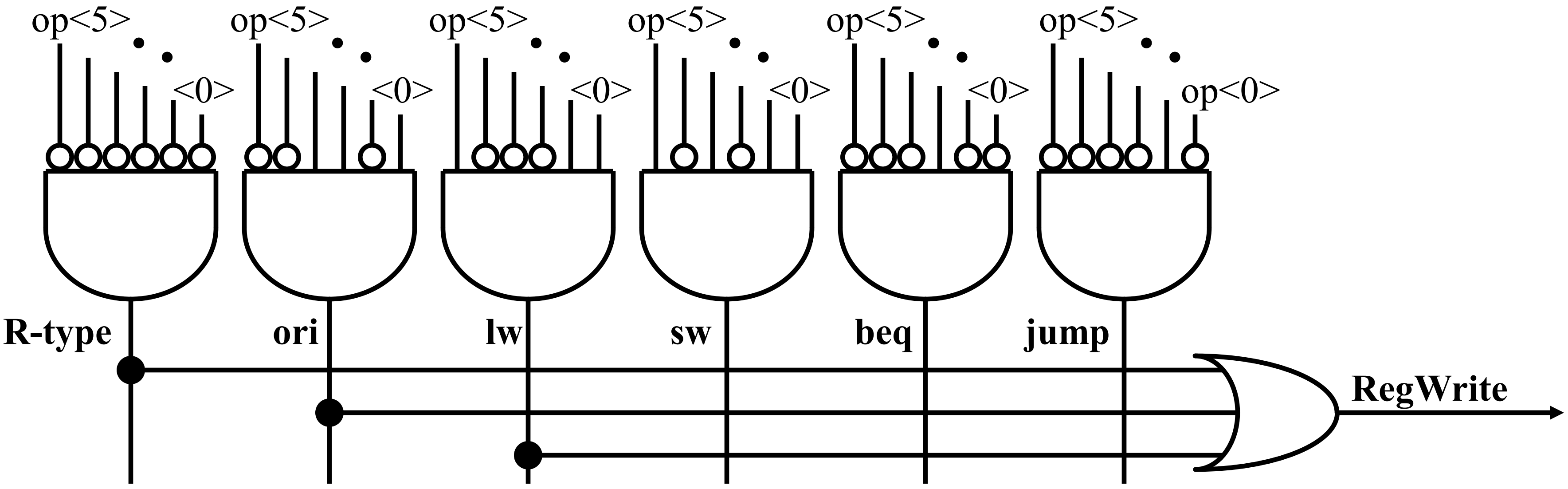
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

- RegWrite = R-type + ori + lw

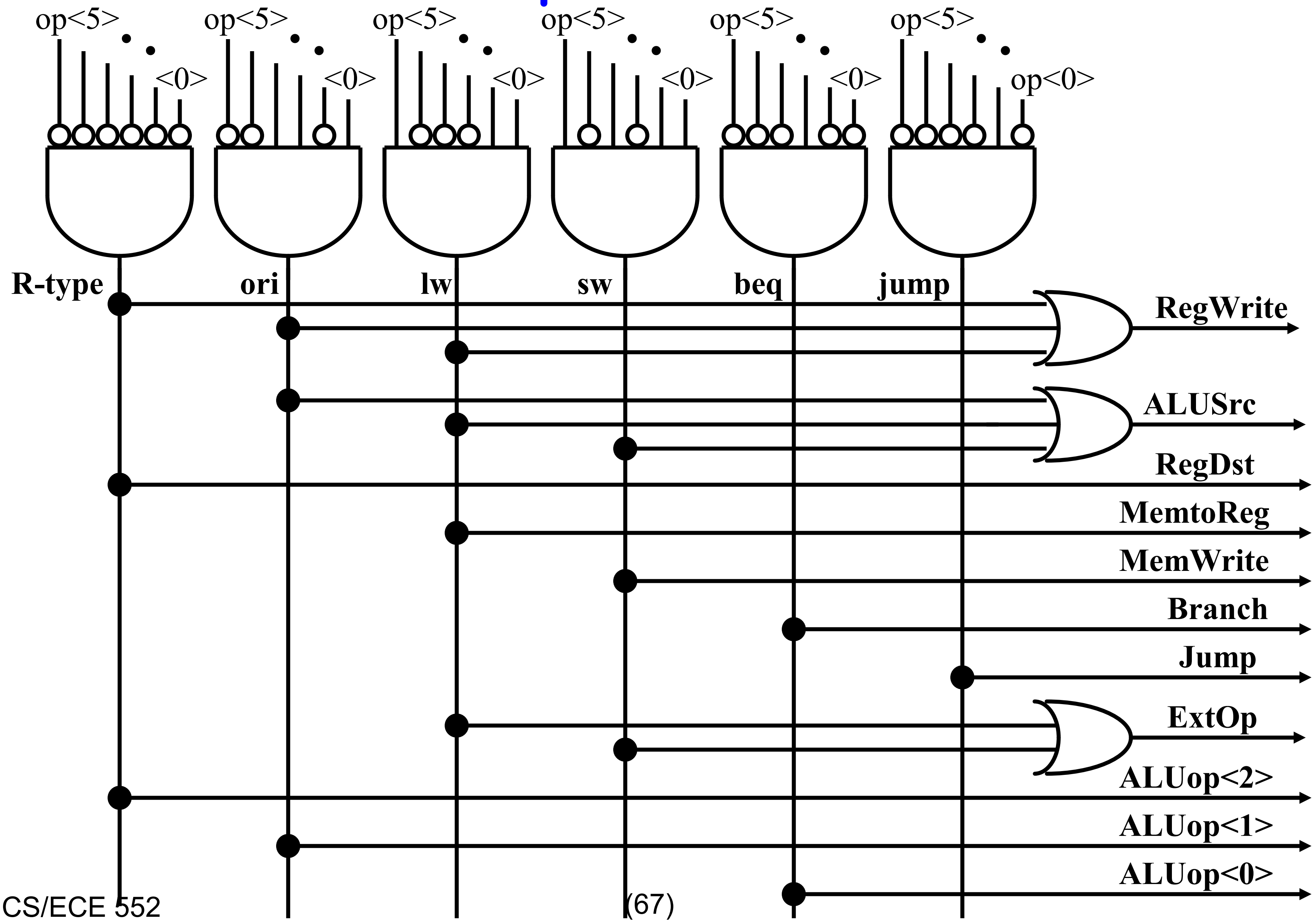
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

  + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)

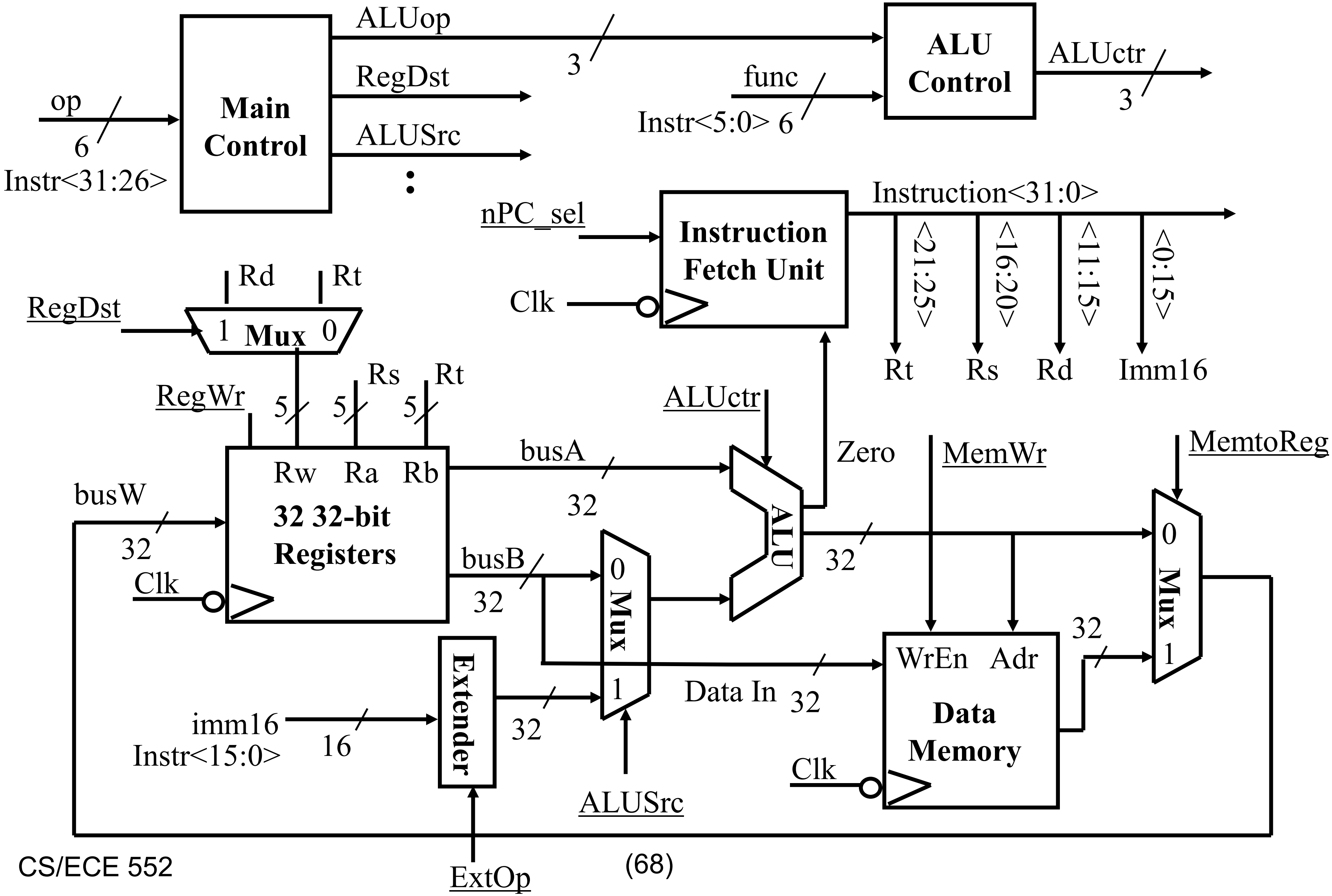
  + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



PLA implementation



Putting it all together



U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

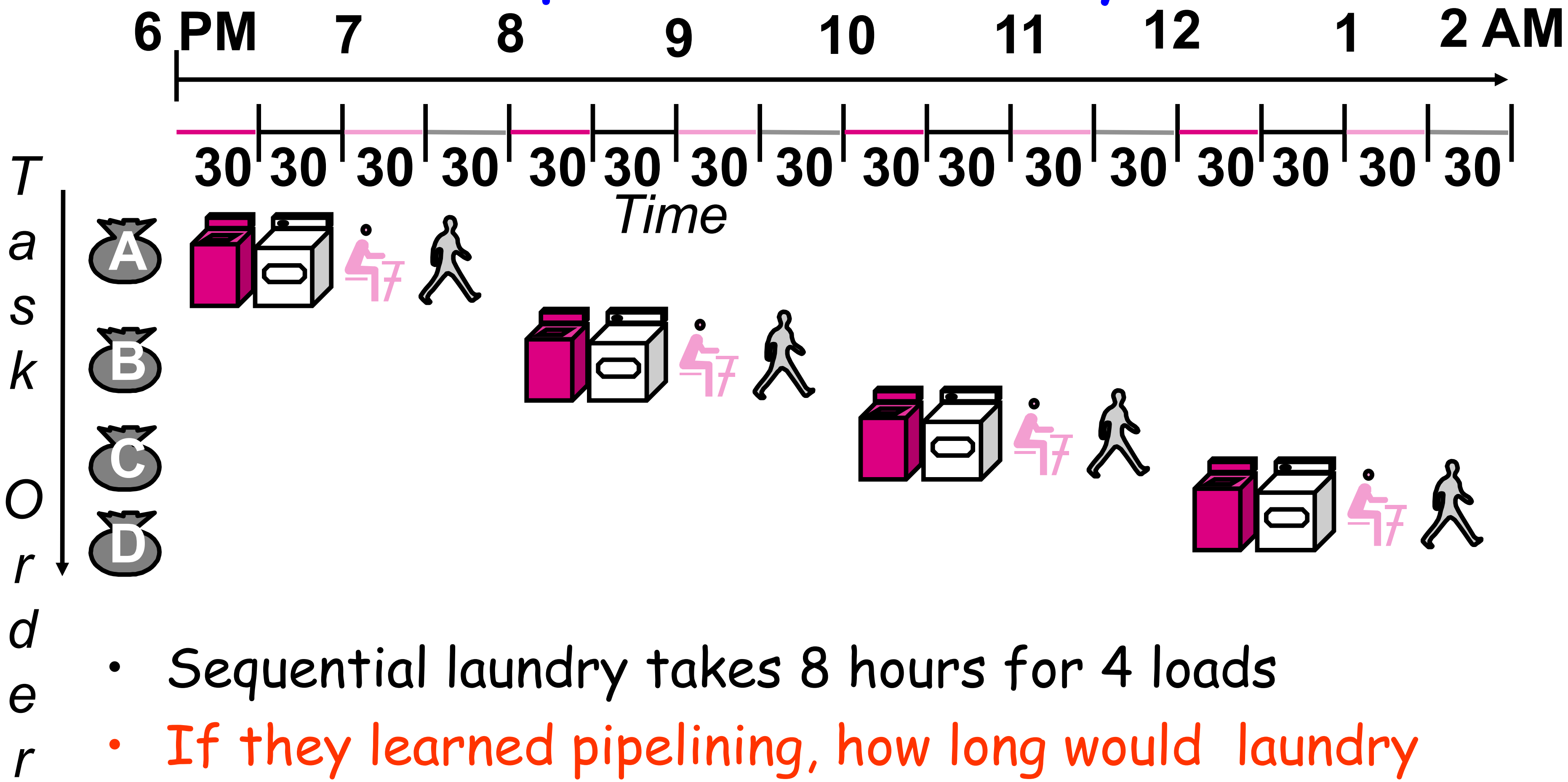
Prof. Karu Sankaralingam

Pipelining (Chapter 6)

www.cs.wisc.edu/~karu/cs552/

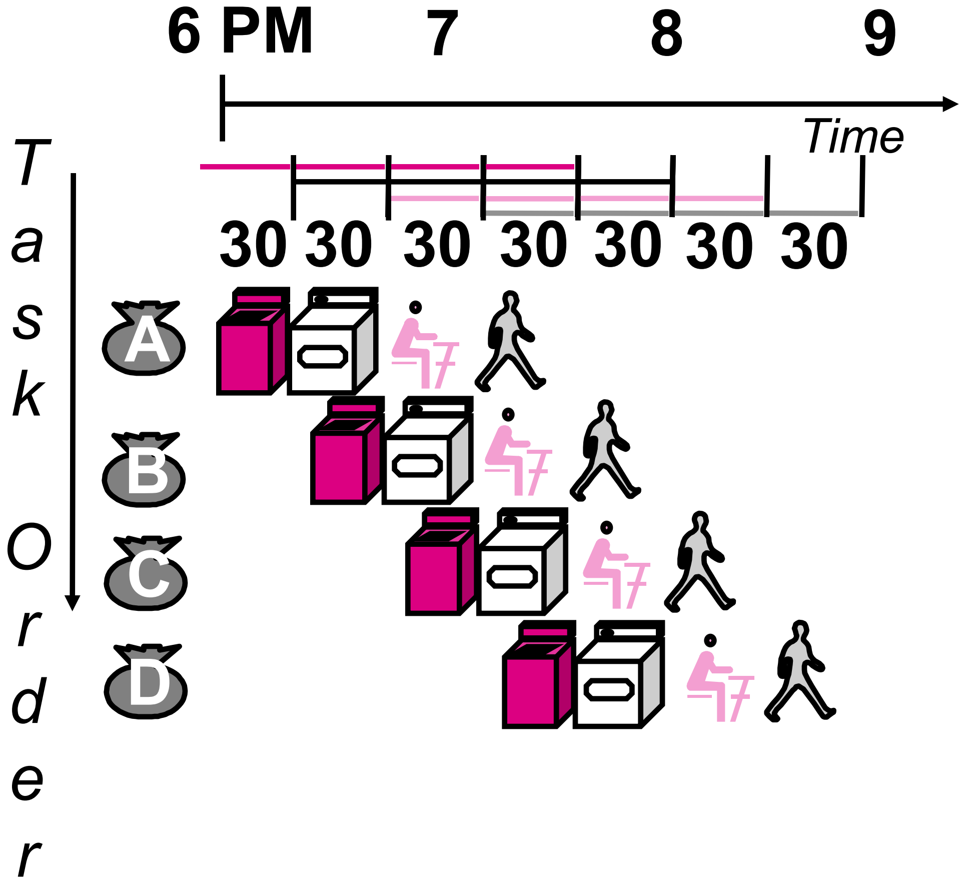
Slides combined and enhanced by Mark D. Hill from work
by Falsafi, Marculescu, Nagle, Patterson, Roth, Rutenbar,
Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Sequential Laundry



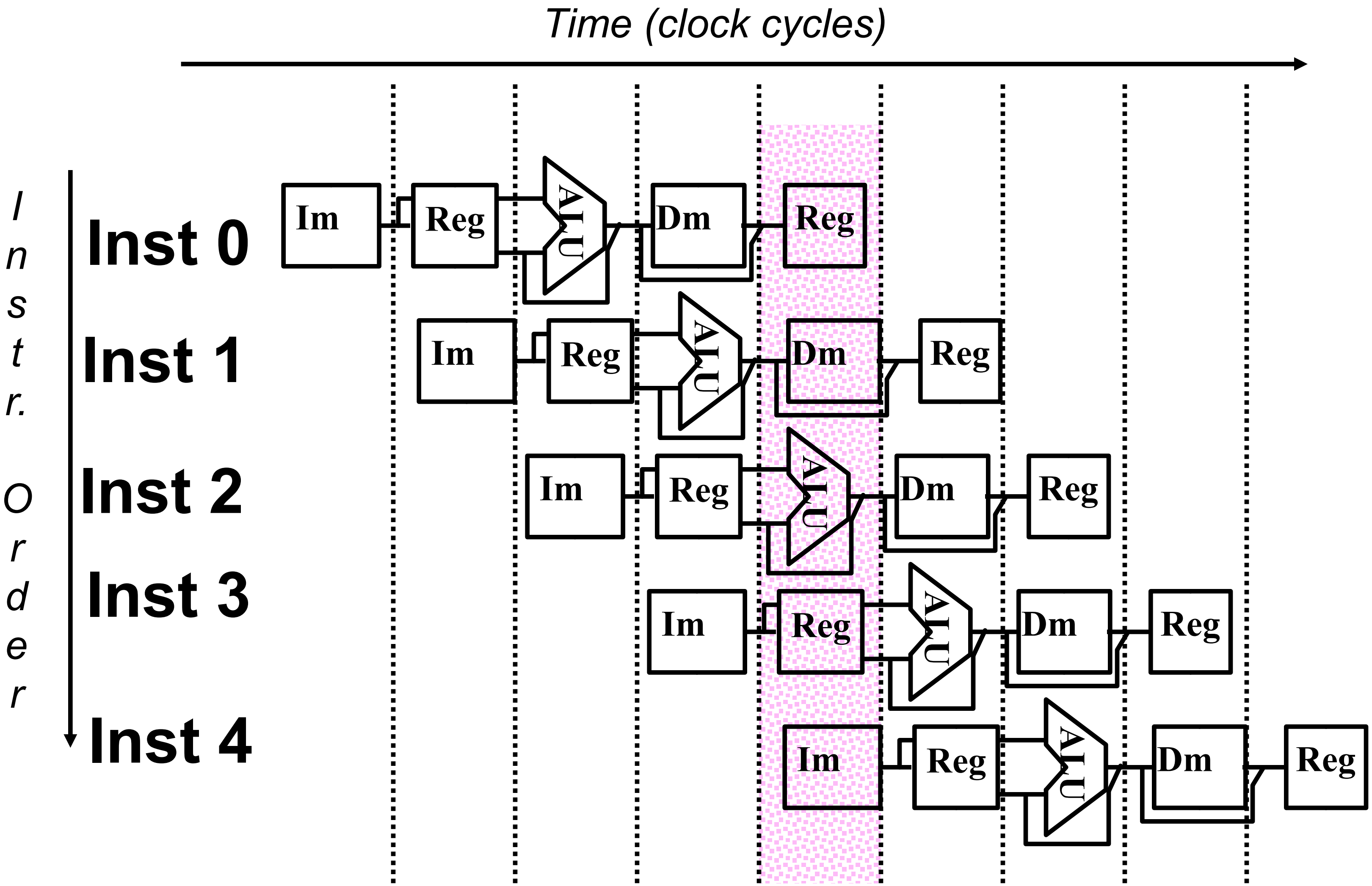
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelining lessons



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

Seek to Pipeline Instructions

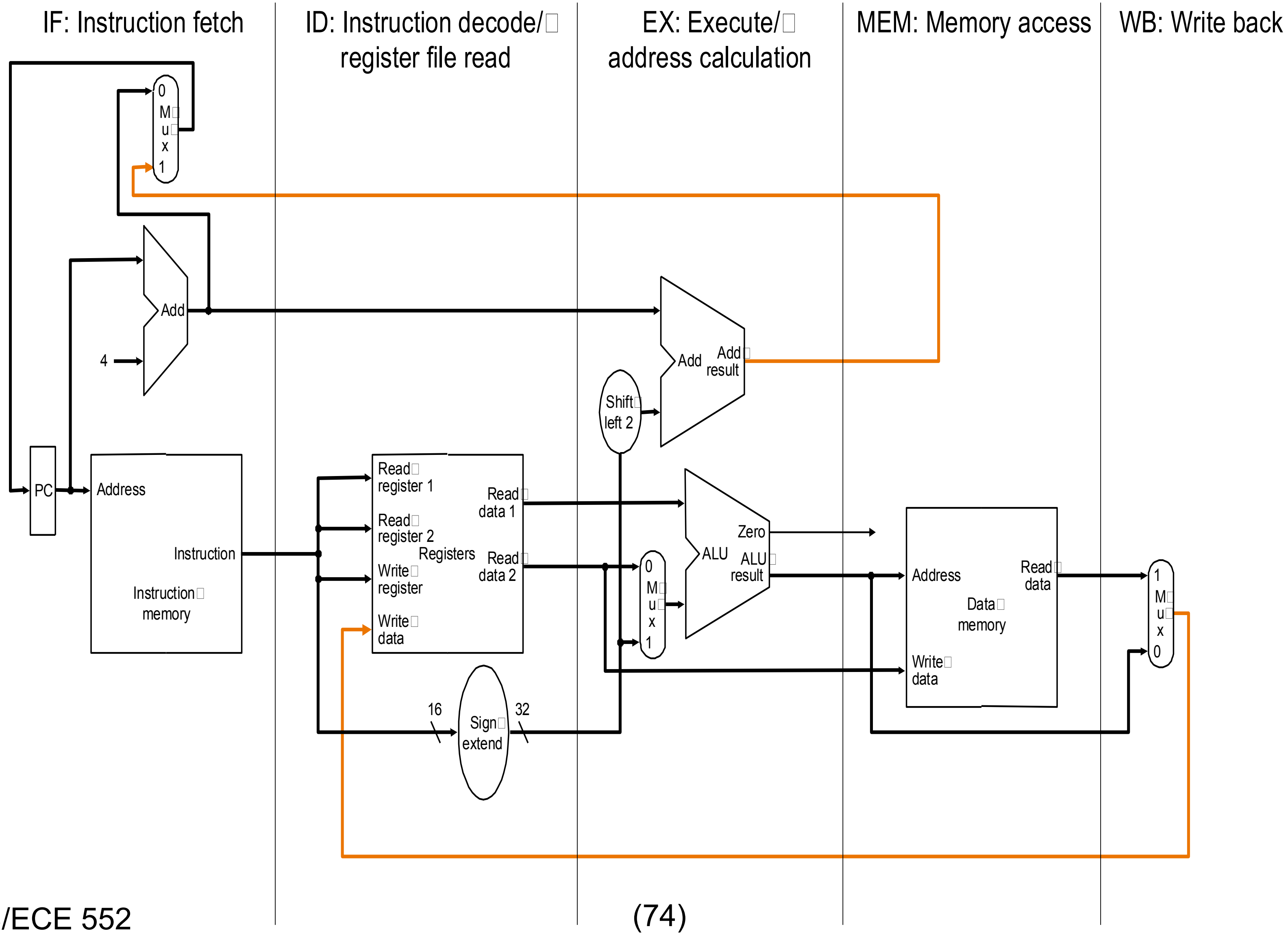


Non-uniform stages

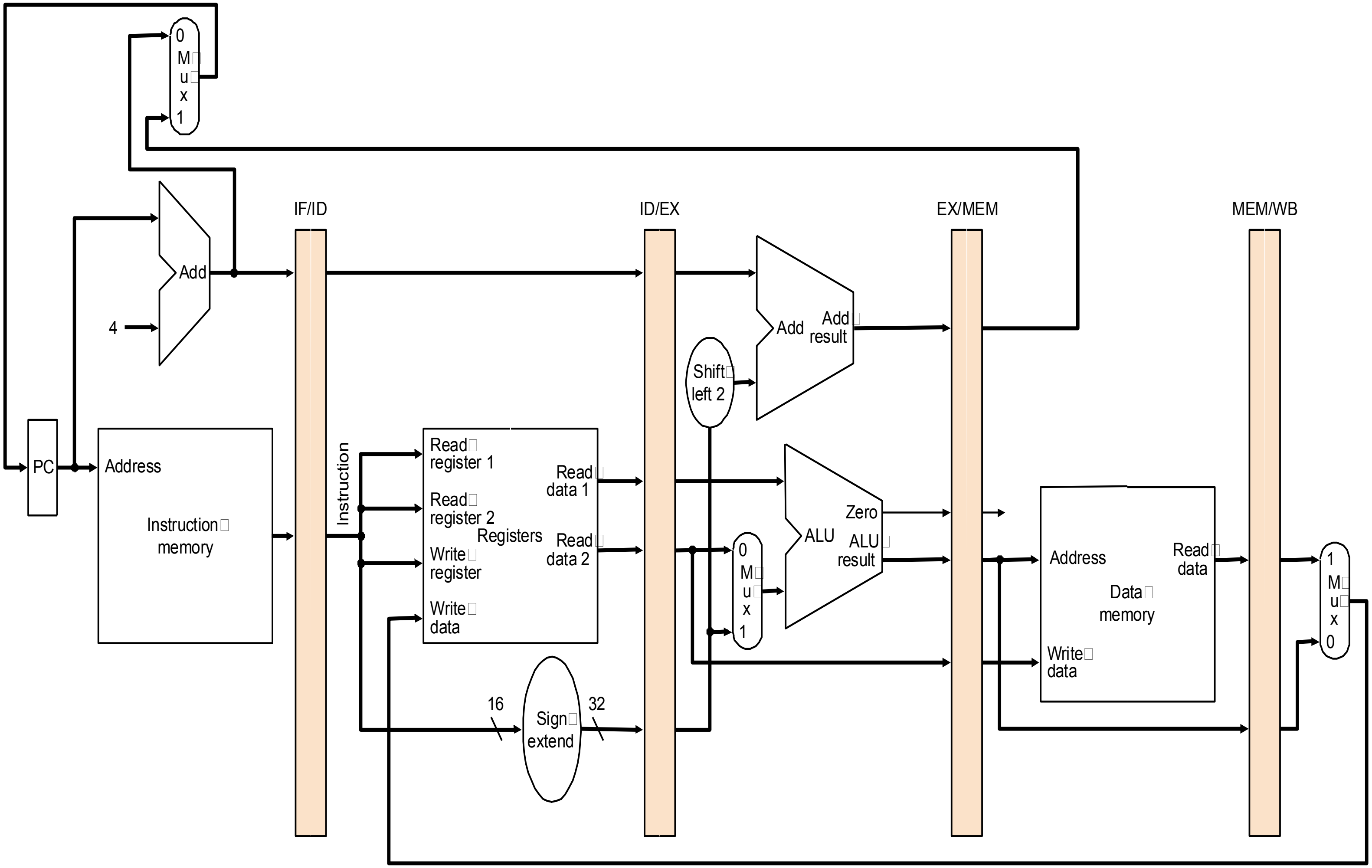
Maximum Speedup $\leq$ Number of stages

$$\text{Speedup} \leq \frac{\text{Time for unpipelined operation}}{\text{Time for longest stage}}$$

Pipeline Forecast: Single-Cycle Datapath

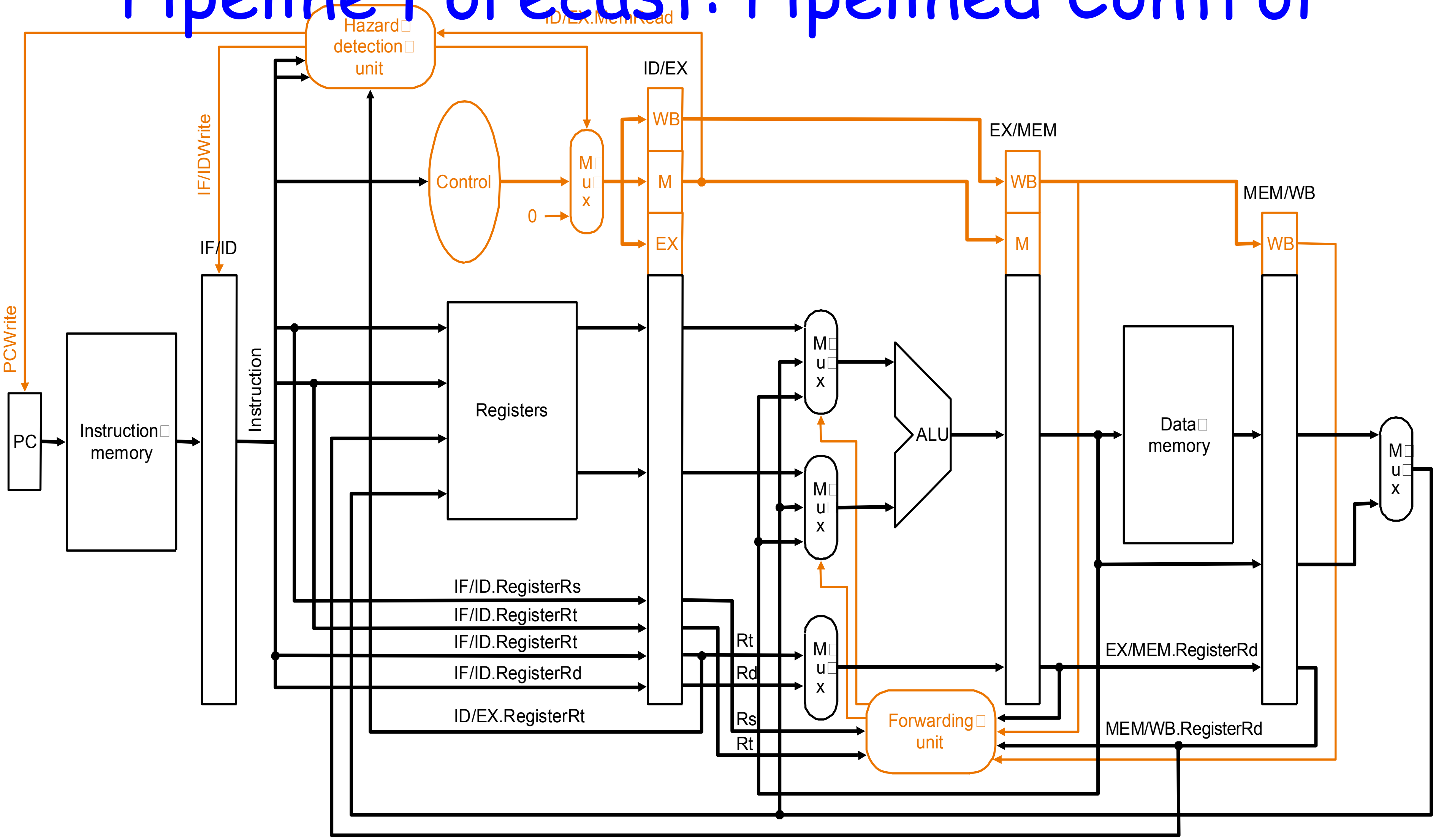


Pipeline Forecast: Pipelined Datapath



- Pipeline datapath with registers

Pipeline Forecast: Pipelined Control



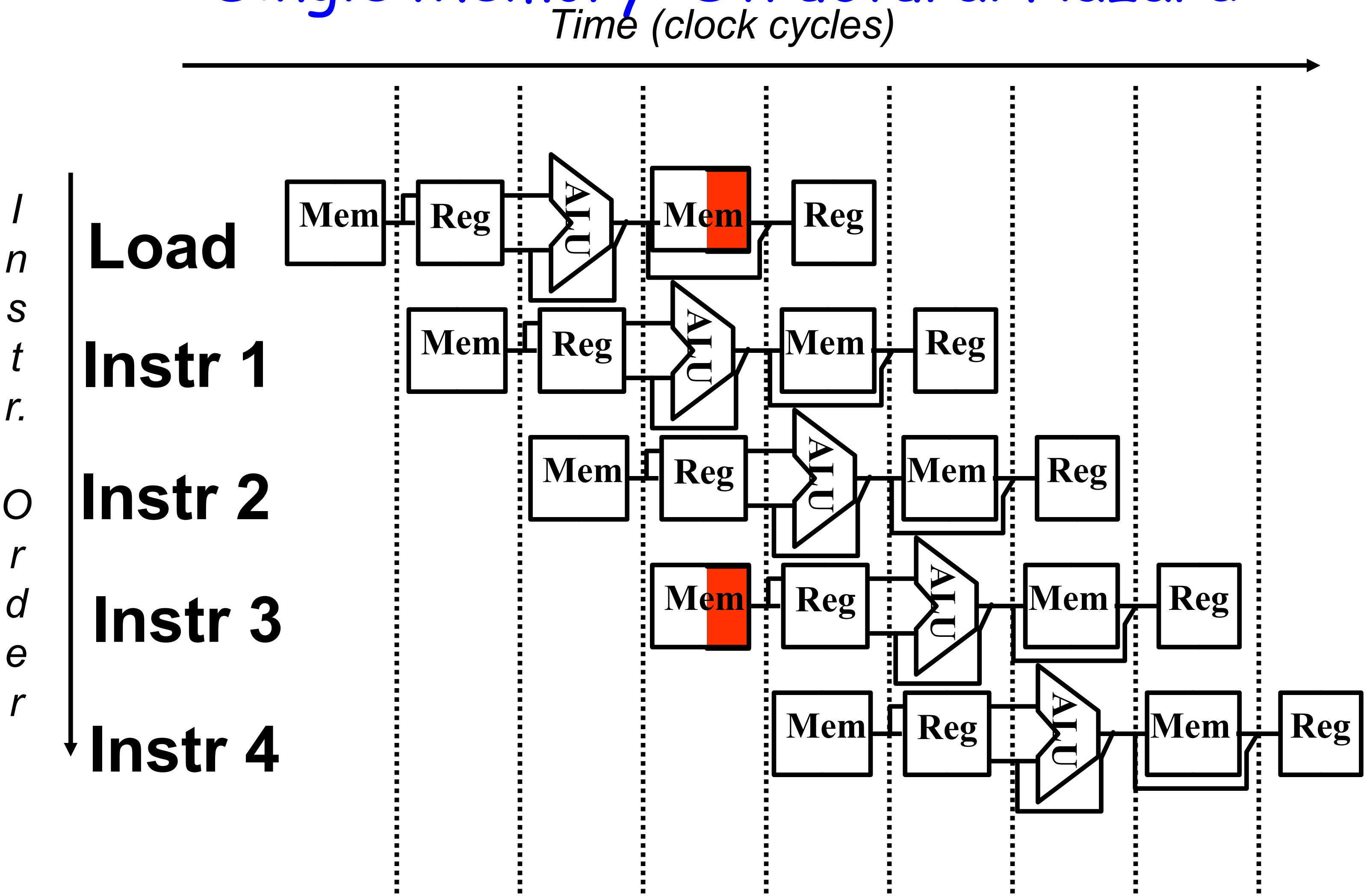
Pipeline Forecast: Big Picture

- Datapath similar to single-cycle datapath
- Partition datapath with pipeline latches (D-FFs)
- Naïve Control
  - Generate single-cycle control signals
  - Pass control signals through pipeline latches
  - Apply control signals at appropriate stage/cycle
- Truth is more complex (instruction interact)

Hazards

- Structural hazards
  - Two instructions need the same hardware
- Data Hazards
  - Data not ready
- Control Hazards
  - Which instruction to fetch? Not known.

Single Memory: Structural Hazard



Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards

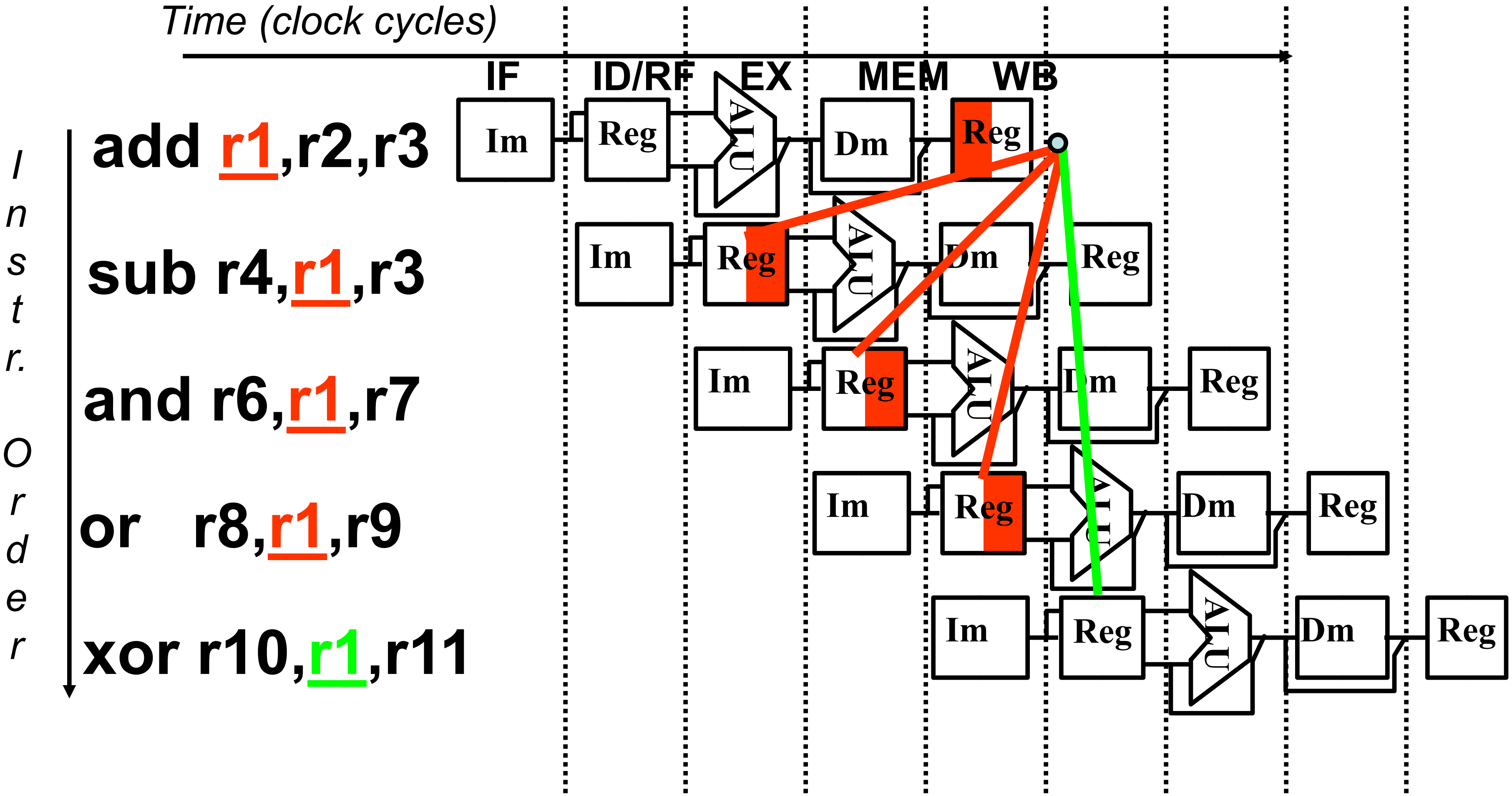
- If 1.3 memory accesses per instruction
  - How?
  - 1 per instruction for instruction fetch
  - Fraction for data load/store
    - Depends on instruction mix
    - 20% load + 10% store
    - 15% load + 15% store
- CPI is atleast 1.3 (otherwise memory is used more than 100%)

Data Hazards

```
add r1 ,r2,r3

sub r4, r1 ,r3

and r6, r1 ,r7

or   r8, r1 ,r9

xor r10, r1 ,r11
```

Hazards on r1

- Dependencies backwards in time



Time (clock cycles)

IF ID/RF EX MEM WB

Instr. Order

add r1,r2,r3

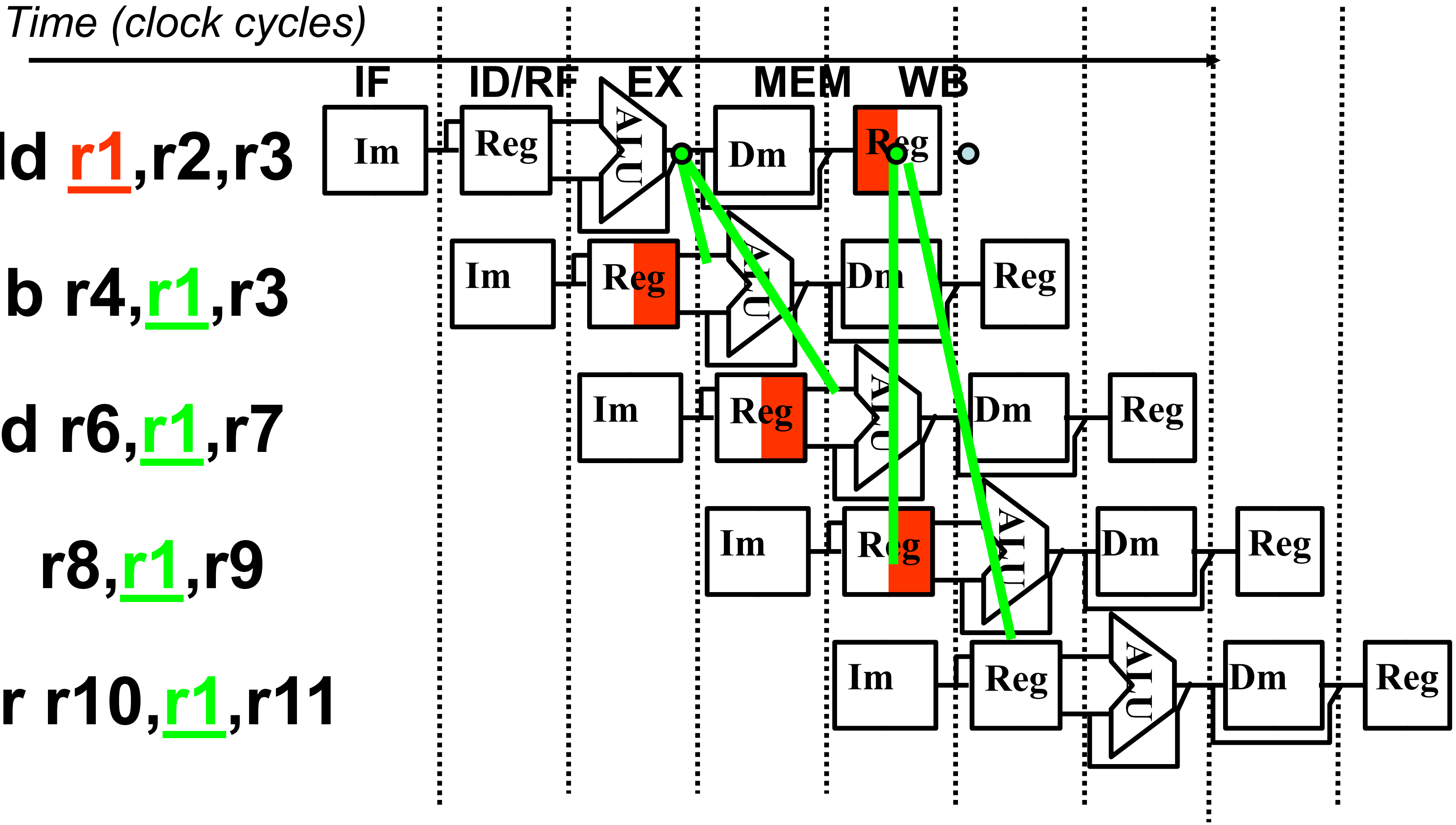
sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11

Data Hazard Solution



add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

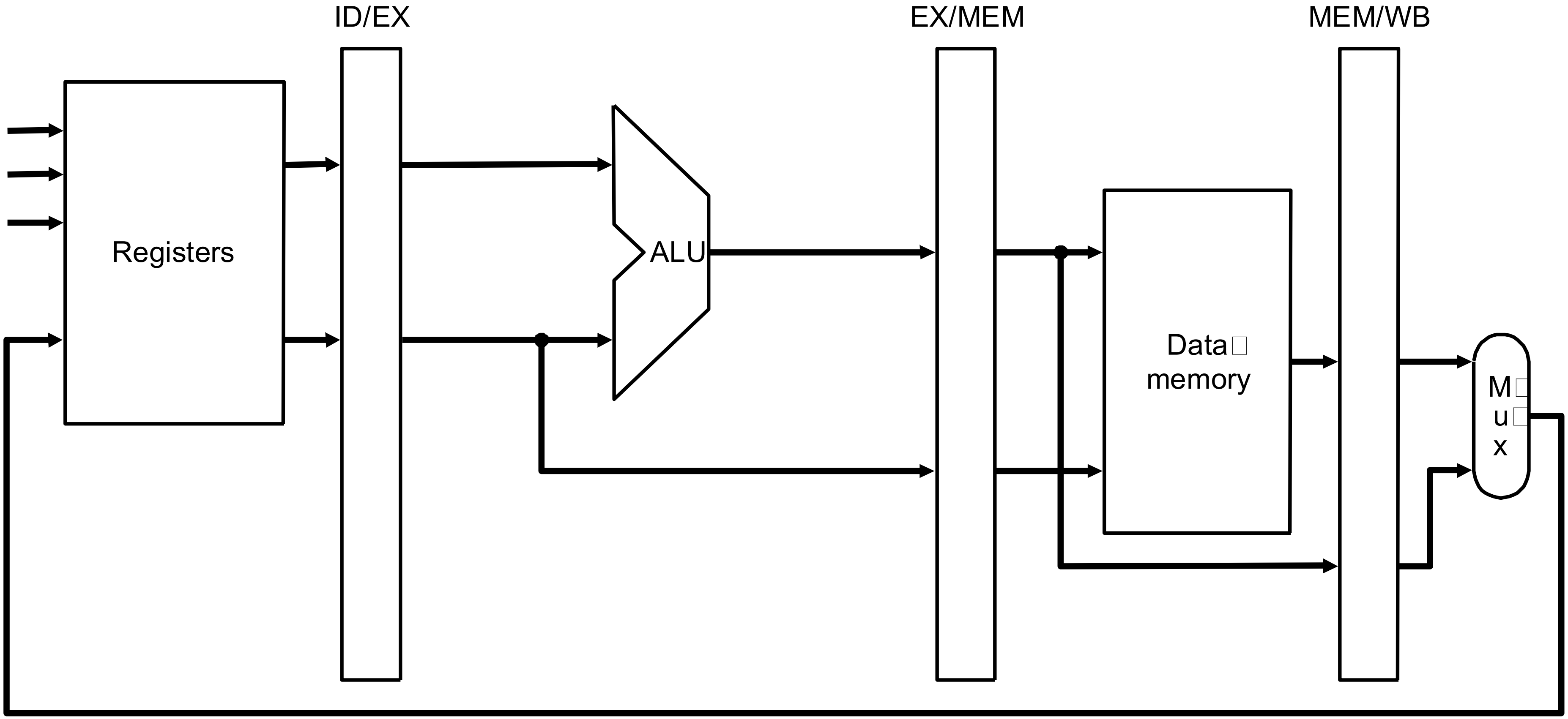
xor r10,r1,r11

Logic equations for Hazard Detection

- Restatement of equations
- Text book version
  - WB stage is not really a hazard
    - Data is written in first half of cycle, read in 2nd half
  - EX/MEM.RegisterRd = ID/EX.RegisterRs
  - EX/MEM.RegisterRd = ID/EX.RegisterRt
  - MEM/WB.RegisterRd = ID/EX.RegisterRs
  - MEM/WB.RegisterRd = ID/EX.RegisterRt

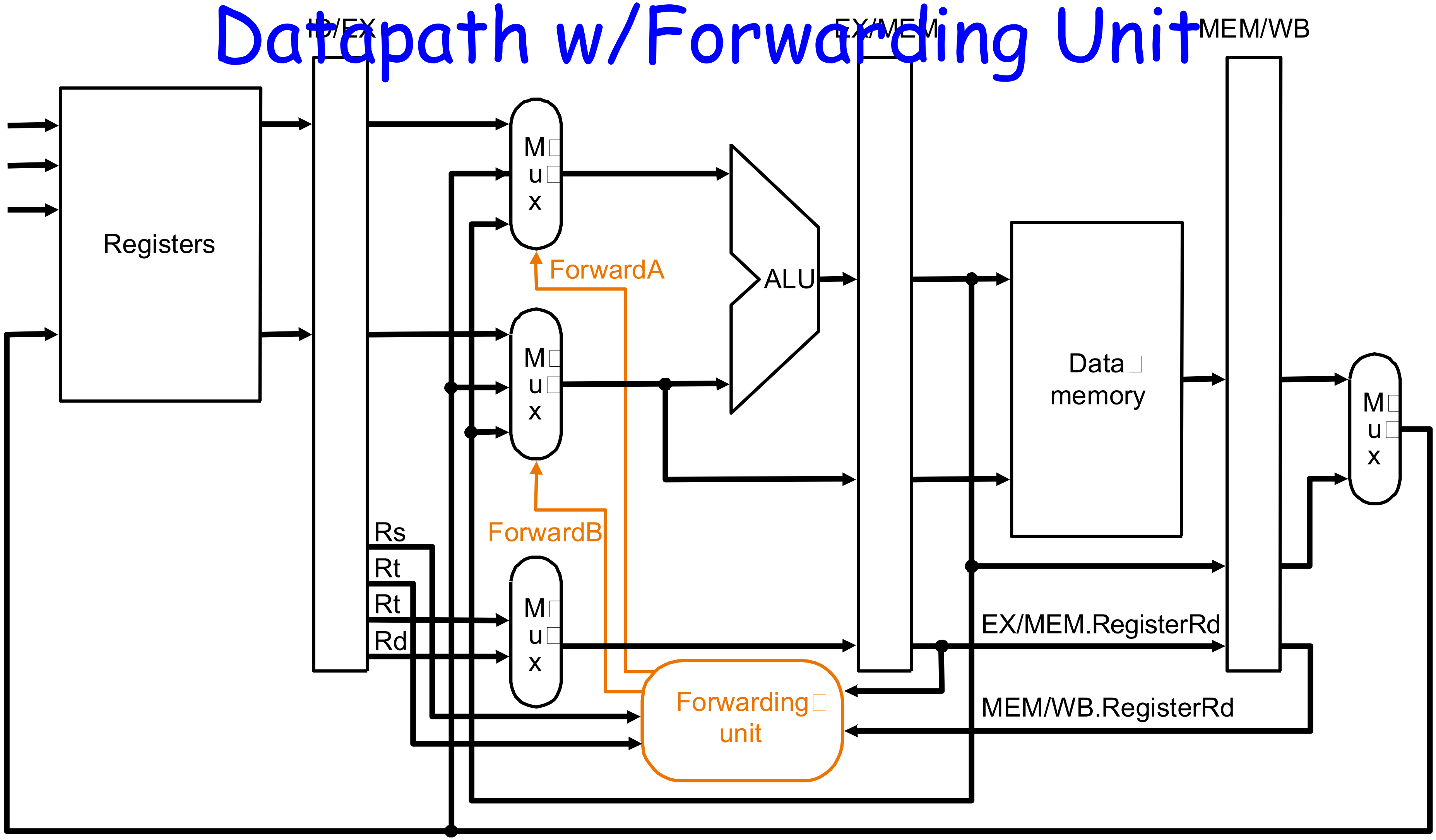
Base Pipelined Datapath

- Simplified representation of pipelined datapath
  - To avoid clutter



a. No forwarding

Datapath w/Forwarding Unit



b. With forwarding

- ForwardA/ForwardB: 01->Mem, 10->EX

Forwarding Control Behavior

- EX hazard

```
If (EX/MEM.RegWrite AND      // not store or branch
        EX/MEM.RegsterRd != 0   AND  // Result is used
        EX/MEM.RegisterRd = ID/EX.RegisterRs)
        ForwardA = 10

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

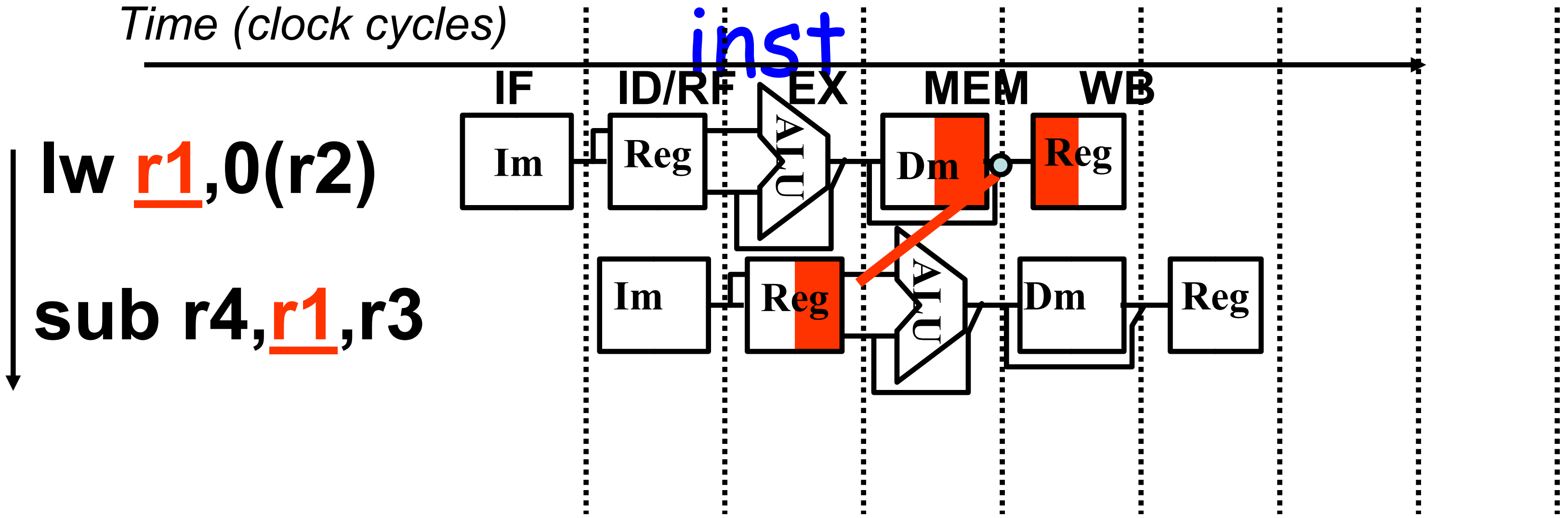
Forwarding Control Behavior

- MEM hazard

```
If (MEM/WB.RegWrite AND
        MEM/WB.RegsterRd != 0   AND
        MEM/WB.RegisterRd = ID/EX.RegisterRs)
        ForwardA = 01

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

- Does this fully meet our requirements ?

Lookahead: RAW hazard with load inst



lw r1,0(r2)

sub r4,r1,r3

- Forwarding as solution to RAW hazard
  - possible if no (true) dependence going backwards in time
  - True for R-type instructions
    - Data available after EX stage (i.e., at ALUOut)
  - Not true for load instruction

Solution

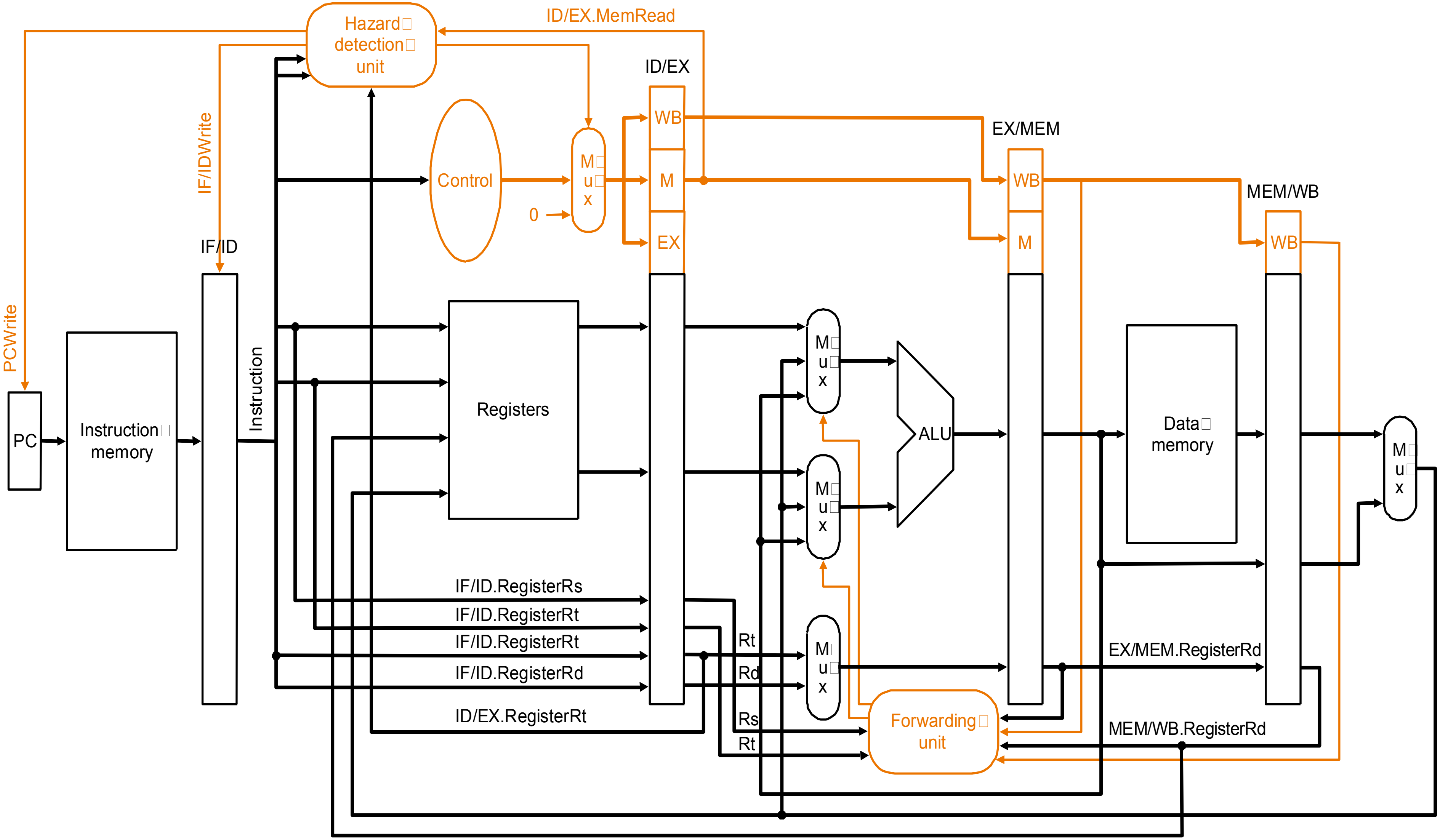
- Catch-all solution for hazards
  - Stall
    - always works, but hurts performance
    - Use as last resort

- Challenge:
  - Modify pipeline implementation to support stalls when hazards are detected

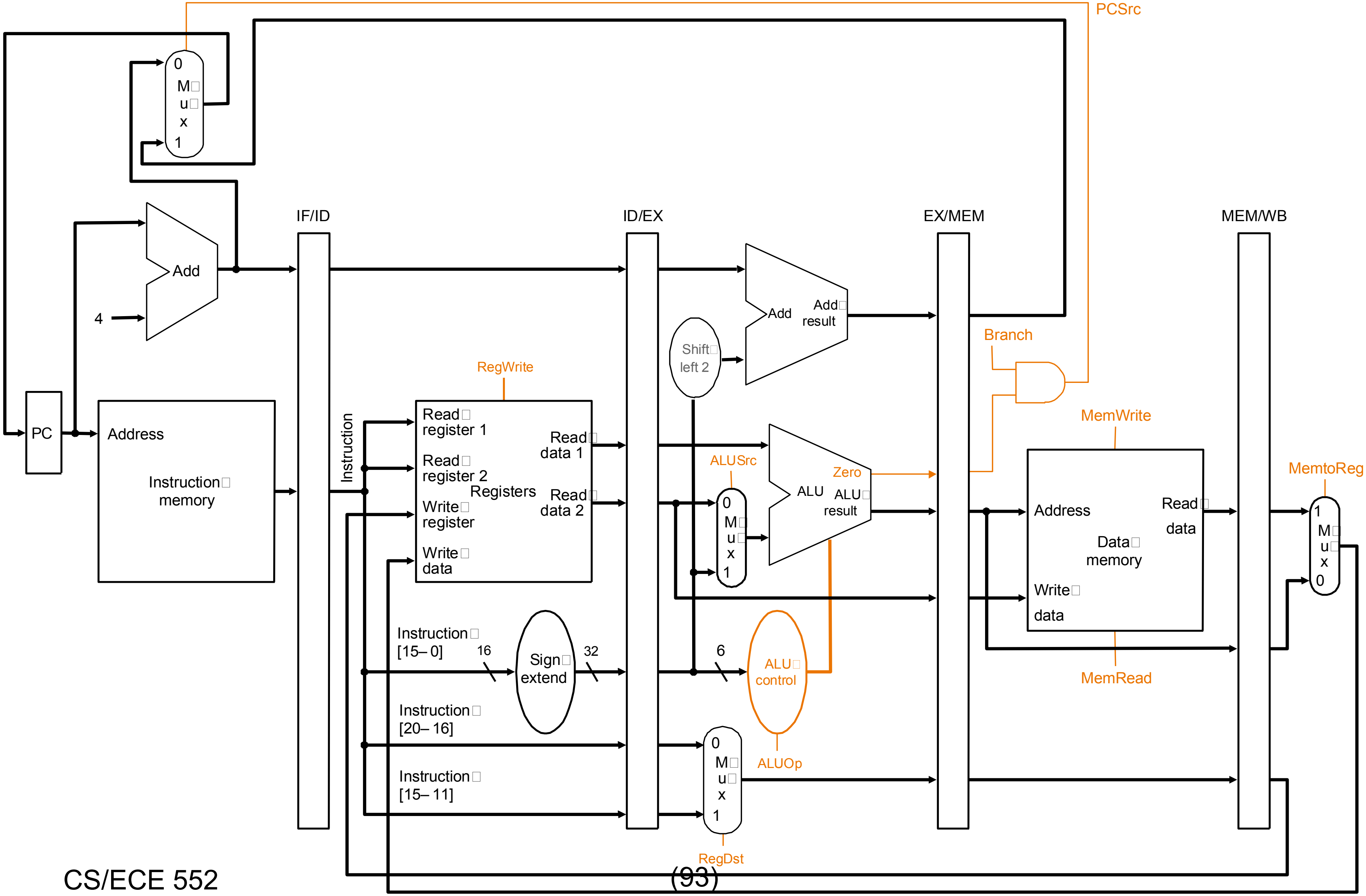
Stalling the pipeline

- Instruction cannot proceed
  - Following instruction must be stalled too.
  - Otherwise state in pipeline registers is overwritten
- Preceding instructions may proceed as usual
- Solution
  - inject NOP into EX/Mem pipeline
  - Prevent writes to PC to IF/ID register

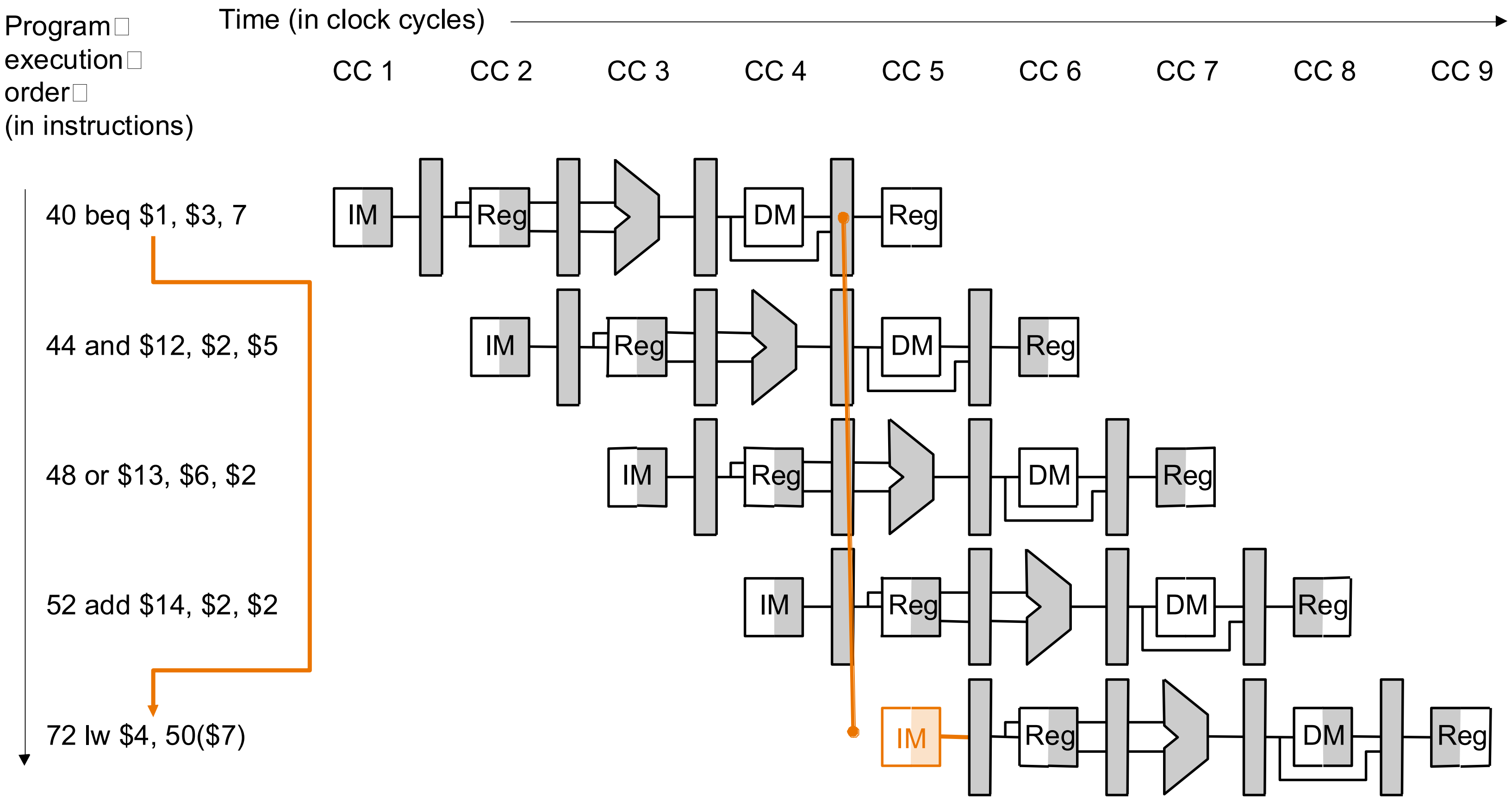
Datapath


Hazard detection unit
ID/EX.MemRead
PCWrite
IF/IDWrite
ID/EX
Control
M u x
0
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
PC
Instruction memory
Instruction
Registers
M u x
M u x
ALU
Data memory
M u x
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
Rt
Rd
M u x
Rs
Rt
ID/EX.RegisterRt
Forwarding unit
EX/MEM.RegisterRd
MEM/WB.RegisterRd

When conditional branches resolved?



Branch Hazards



Program execution order (in instructions)

Time (in clock cycles)

CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9

40 beq $1, $3, 7

44 and $12, $2, $5

48 or $13, $6, $2

52 add $14, $2, $2

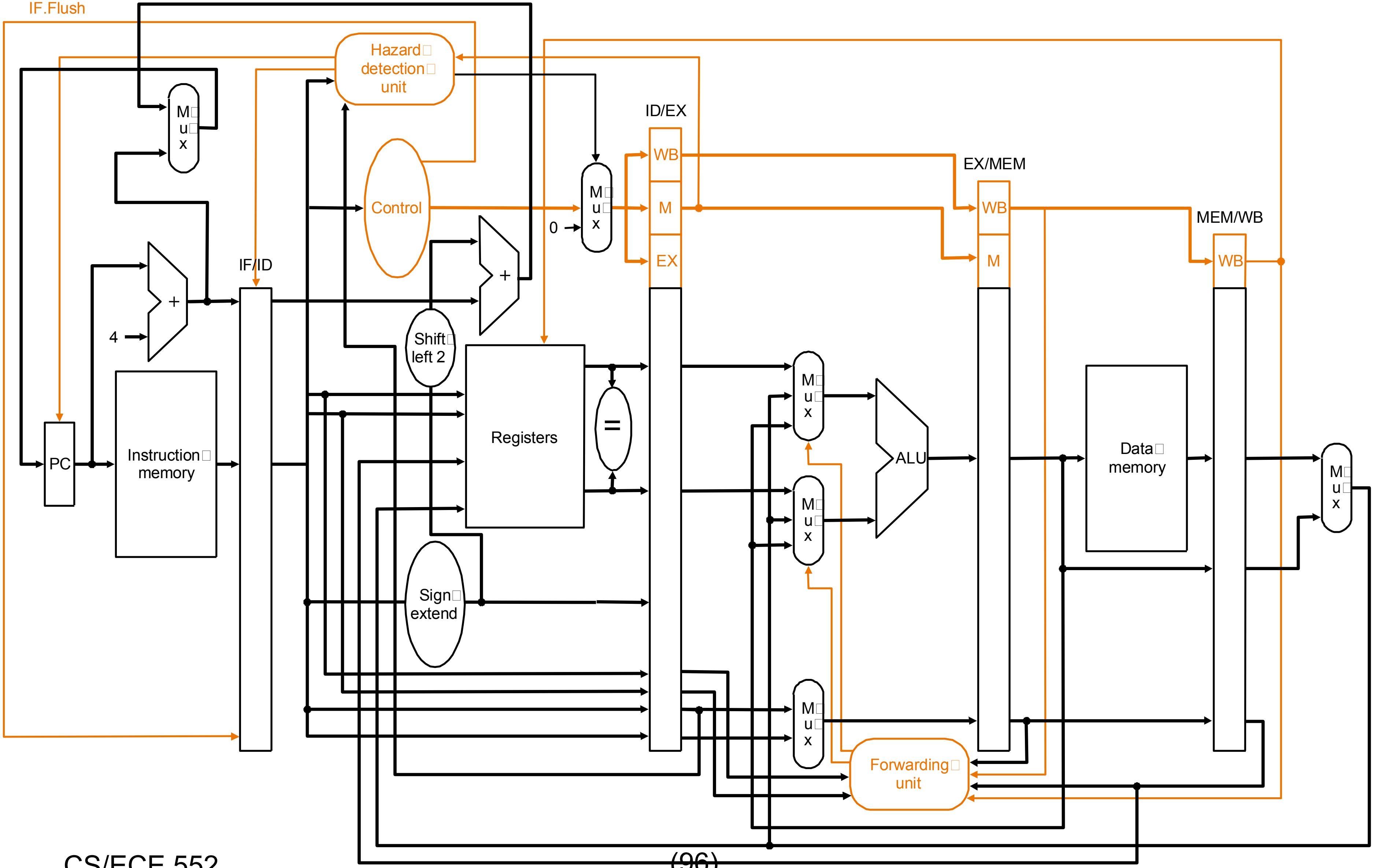
72 lw $4, 50($7)

- Branch resolved in the MEM stage
- If taken,
  - PC<- PC + 4 + SX(Imm*4)
  - 40 + 4 + 7*4 = 72

Control/Branch Hazards

- Branch resolved in the MEM stage
  - But next instruction has to fetched in the next cycle
  - Reduce the penalty by moving decision earlier in pipeline
    - Need additional comparator (r1=r2?) and adder (PC+4+SX(IMM)*4)
  - Reduced penalty from 3 cycles to 1 cycle

Datapath for branch hazards



Eliminate 1-cycle stall?

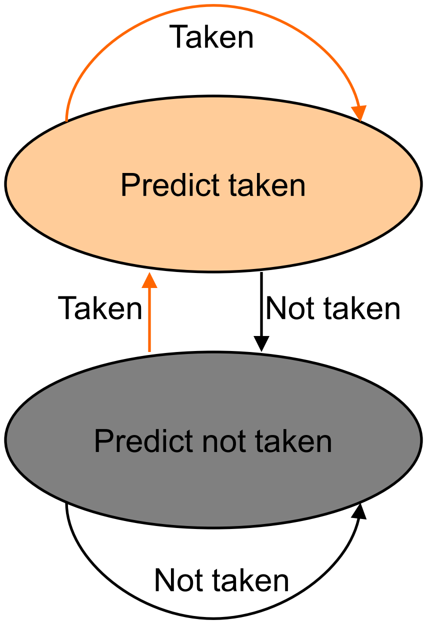
- Two solutions
  - Predict branch is always not taken
    - More sophisticated prediction schemes
  - Delay slots
    - Compiler's problem

- Walkthrough example for solution #1
  - Predict not taken

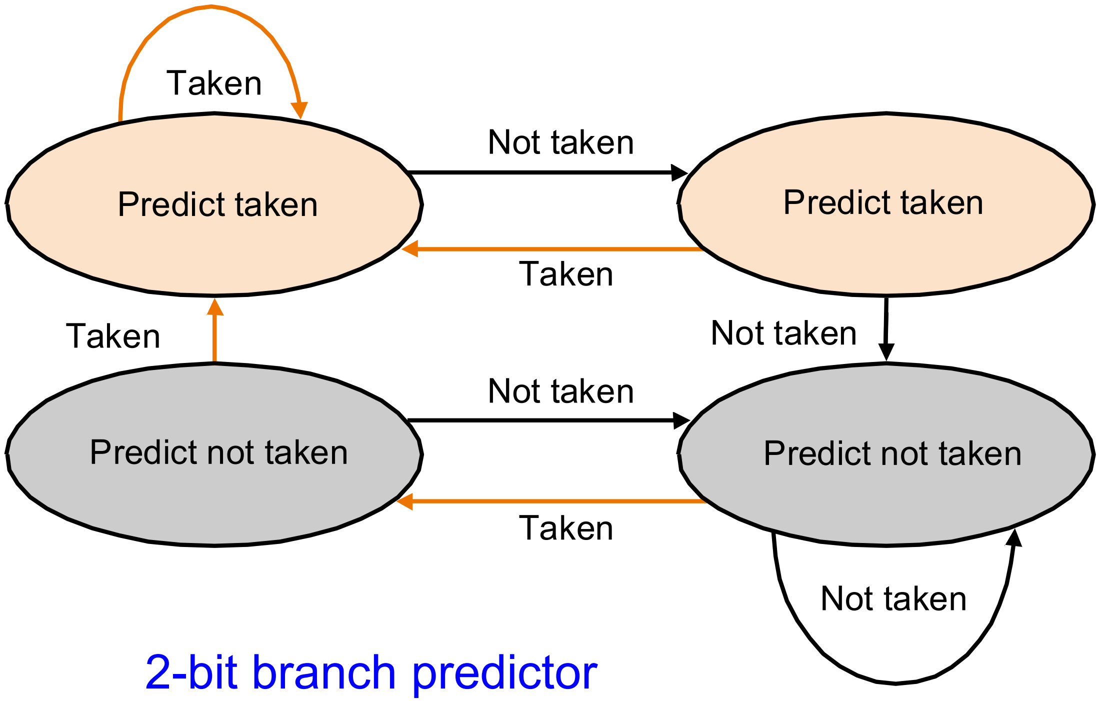
Dynamic Branch Prediction

- Better than static prediction
  - Branches are predictable
  - ~90% of program execution time is spent in ~10% of code (inner loops)
  - Think of a program loop of N iterations
    - Taken N-1 times
    - Not taken last time

Dynamic Branch Prediction



1-bit branch predictor

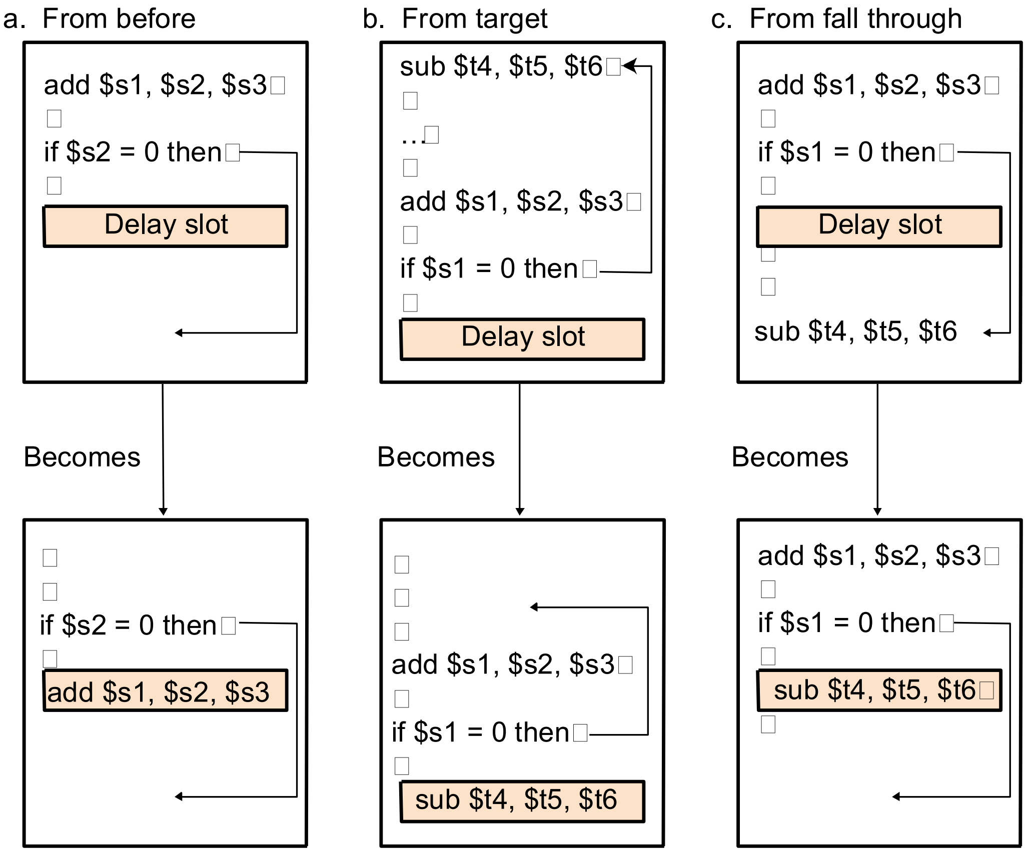


2-bit branch predictor

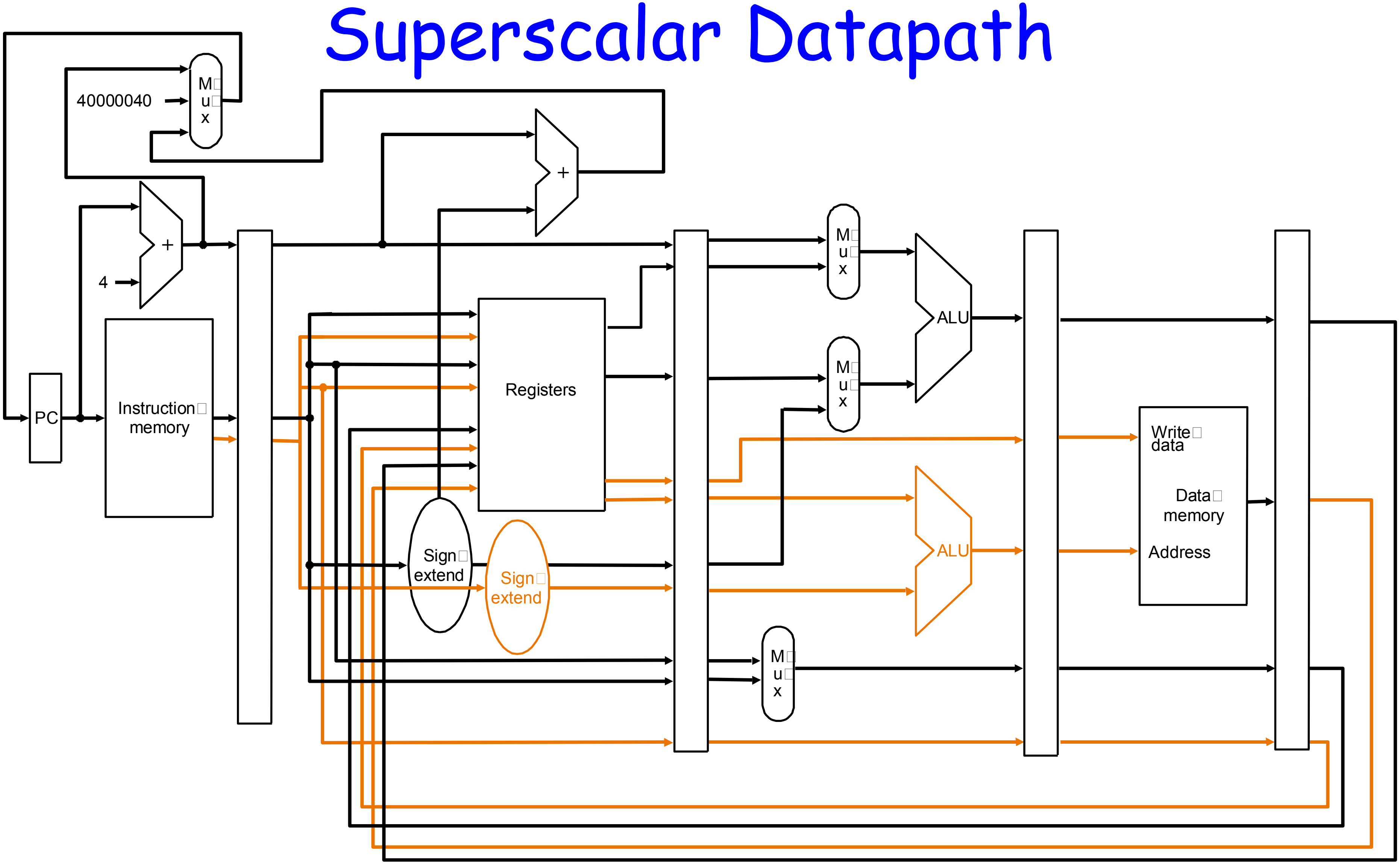
- How does hardware “learn” branch behavior?
- Store each branch instruction’s history ***
  - If a branch was taken “recently”, predict taken
    - One bit saturating counter
    - Two bit counters

“Easy way”* to hide branch hazard delay

- Delayed branch
  - Instruction after branch always executes
  - Find an independent instruction from before the branch
  - Find instructions from Taken (target) OR from Not Taken (fall-through) code section
- * For Architects



Superscalar Datapath



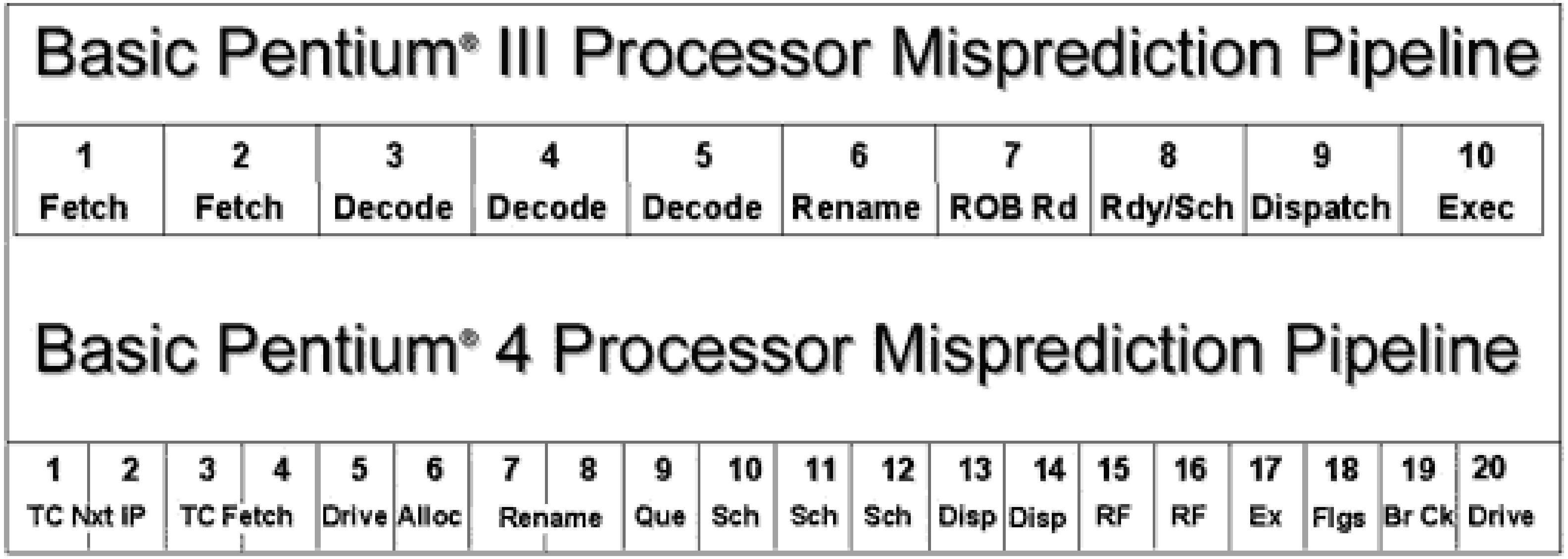
- Replicate datapath elements

Dynamic Scheduling

- No need to suffer hazards if other useful work can be achieved
- Load Hazard results in pipeline stall
  - But other instructions are ready
  - "Oh! But we cannot execute instructions out of order" – Not really

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, $t3
```

Pentium 4 pipeline



Basic Pentium® III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium® 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

- Pipeline too much; c.f., Core2