inst.eecs.berkeley.edu/~cs61c/su06

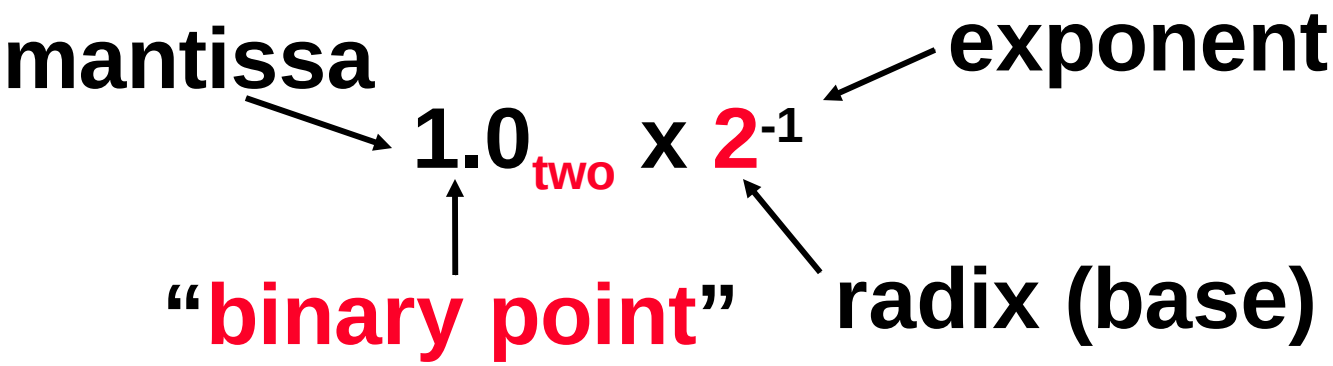
CS61C : Machine Structures

Lecture #20: Midterm 2 Review

Midterm 2: Friday 11-2
390 Hearst Mining

2006-08-01

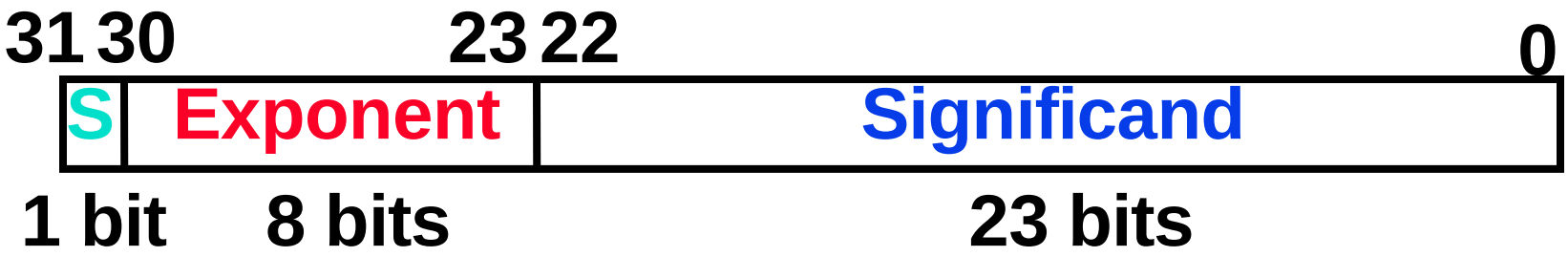
Scientific Notation (in Binary)



- Normalized mantissa always has exactly one "1" before the point.
- Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers
- Declare such variable in C as `float`

Floating Point Representation

- **Normal format: +1.xxxxxxxxxx$_{two}$*2^{yyyy}$_{two}$**
- **Multiple of Word Size (32 bits):**

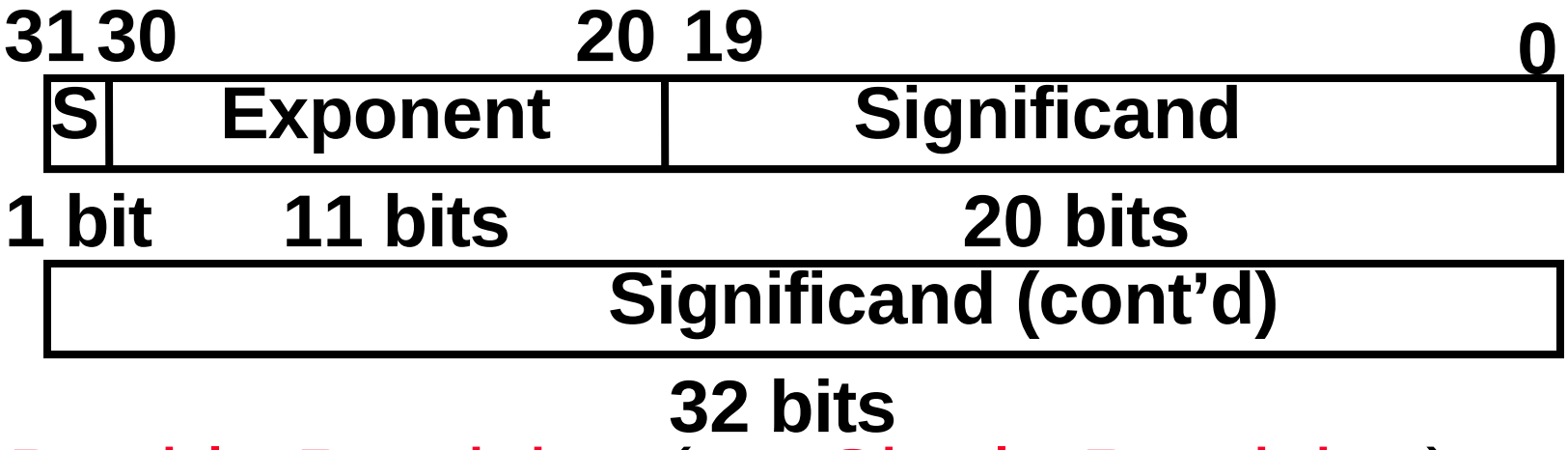


- S represents Sign
- Exponent represents y's
- Significand represents x's

Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

Double Precision Fl. Pt. Representation

- **Next Multiple of Word Size (64 bits)**

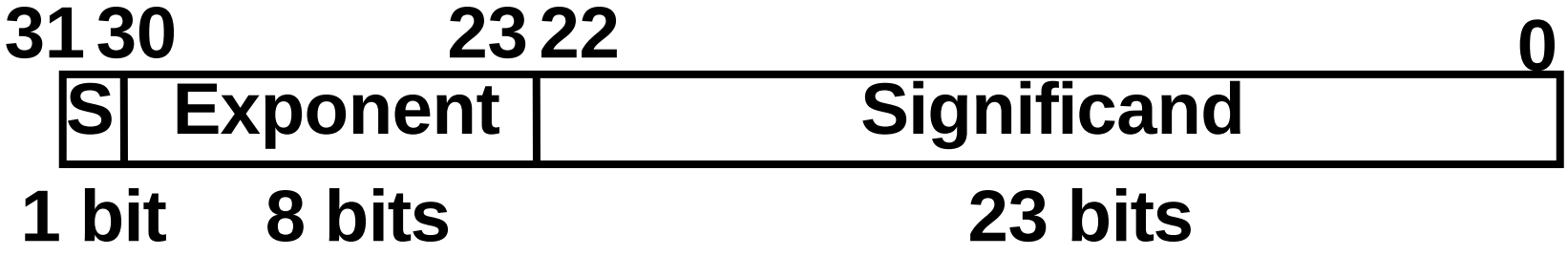


- Double Precision (vs. Single Precision)
  - C variable declared as `double`
  - Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}
  - But primary advantage is greater accuracy due to larger significand

IEEE 754 Floating Point Standard

- Called Biased Notation, where bias is number subtracted to get real number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get actual value for exponent

- **Summary (single precision):**



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - Double precision identical, except with exponent bias of 1023

Representation for ± ∞

- In FP, divide by 0 should produce ± ∞, not overflow.
- Why?
  - OK to do further computations with ∞ E.g., X/0 > Y may be a valid comparison
  - Ask math majors
- IEEE 754 represents ± ∞
  - Most positive exponent reserved for ∞
  - Significands all zeroes

Representation for 0

- Represent 0?
  - exponent all zeroes
  - significand all zeroes
  - What about sign?
  - `+0: 0 00000000 00000000000000000000000`
  - `-0: 1 00000000 00000000000000000000000`
- Why two zeroes?
  - Helps in some limit comparisons
  - Ask math majors

Representation for Not a Number

- What is `sqrt(-4.0)` or `0/0`?
  - If ∞ not an error, these shouldn't be either.
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN,X) = NaN

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable pos num:

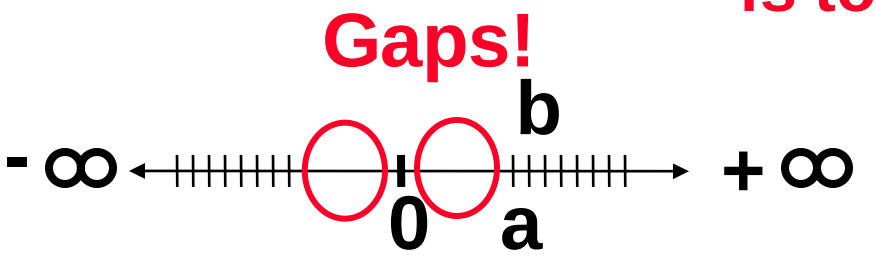
    $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable pos num:

    $b = 1.000\ldots\ldots1_2 * 2^{-126} = 2^{-126} + 2^{-149}$

$a - 0 = 2^{-126}$

$b - a = 2^{-149}$

Normalization and implicit 1 is to blame!

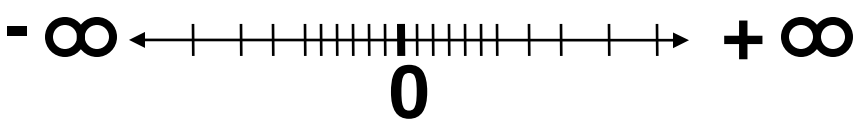


Representation for Denorms (2/2)

- Solution:
  - We still haven't used Exponent = 0, Significand nonzero
  - Denormalized number: no leading 1, implicit exponent = -126.
  - Smallest representable pos num:

    $a = 2^{-149}$
  - Second smallest representable pos num:

    $b = 2^{-148}$



IEEE Four Rounding Modes

- Round towards + ∞
  - ALWAYS round "up": 2.1 ⇒ 3, -2.1 ⇒ -2
- Round towards - ∞
  - ALWAYS round "down": 1.9 ⇒ 1, -1.9 ⇒ -2
- Truncate
  - Just drop the last bits (round towards 0)
- Round to (nearest) even (default)
  - Normal rounding, almost: 2.5 ⇒ 2, 3.5 ⇒ 4
  - Like you learned in grade school
  - Insures fairness on calculation
  - Half the time we round up, other half down

Integer Multiplication

- **Example:**
  - **in C:** `a = b * c;`
  - **in MIPS:**
    - let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)

```
mult $s2,$s3   # b*c          mfhi
$s0   # upper half of
   # product into $s0
mflo $s1      # lower half of
                  # product into $s1
```

- **Note: Often, we only care about the lower half of the product.**

Integer Division

- Syntax of Division (signed):
  - `div` register1, register2
  - Divides 32-bit register 1 by 32-bit register 2:
  - puts remainder of division in `hi`, quotient in `lo`
- Implements C division (/) and modulo (%)
- Example in C: `a = c / d;`
  `b = c % d;`
- in MIPS: `a↔$s0;b↔$s1;c↔$s2;d↔$s3`

```
div  $s2,$s3  # lo=c/d, hi=c%d      mflo
$s0    # get quotient
mfhi $s1     # get remainder
```

Unsigned Instructions & Overflow

- MIPS also has versions of `mult`, `div` for unsigned operands:

  `multu`

  `divu`

  - Determines whether or not the product and quotient are changed if the operands are signed or unsigned.

- MIPS does not check overflow on ANY signed/unsigned multiply, divide instr
  - Up to the software to check `hi`

FP Addition & Subtraction

- Much more difficult than with integers (can't just add significands)
- How do we do it?
  - De-normalize to match larger exponent
  - Add significands to get resulting one
  - Normalize (& check for under/overflow)
  - Round if needed (may need to renormalize)
- If signs ≠, do a subtract. (Subtract similar)
  - If signs ≠ for add (or = for sub), what's ans sign?

MIPS Floating Point Architecture

- Separate floating point instructions:
  - Single Precision: `add.s`, `sub.s`, `mul.s`, `div.s`
  - Double Precision: `add.d`, `sub.d`, `mul.d`, `div.d`
- These are <u>far more complicated</u> than their integer counterparts
  - Can take much longer to execute

MIPS Floating Point Architecture

- 1990 Solution: Make a completely separate chip that handles only FP.
- Coprocessor 1: FP chip
  - contains 32 32-bit registers: `$f0`, `$f1`, …
  - most of the registers specified in `.s` and `.d` instruction refer to this set
  - separate load and store: `lwc1` and `swc1` ("load word coprocessor 1", "store …")
  - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`, … , `$f30`/`$f31`
    - Even register is the name

FP/Math Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

- Integer `mult`, `div` uses `hi`, `lo` regs
  - `mfhi` and `mflo` copies out.
- Four rounding modes (to even default)
- MIPS FL ops complicated, expensive

True Assembly Language

- Pseudoinstruction: A MIPS instruction that doesn't turn directly into a machine language instruction, but into other MIPS instructions
- What happens with pseudoinstructions?
  - They're broken up by the assembler into several "real" MIPS instructions.
  - But what is a "real" MIPS instruction? Answer in a few slides
- First some examples

Example Pseudoinstructions

- **Register Move**

  `move reg2,reg1`

  Expands to:

  `add reg2,$zero,reg1`

- **Load Immediate**

  `li reg,value`

  If value fits in 16 bits:

  `addi reg,$zero,value`

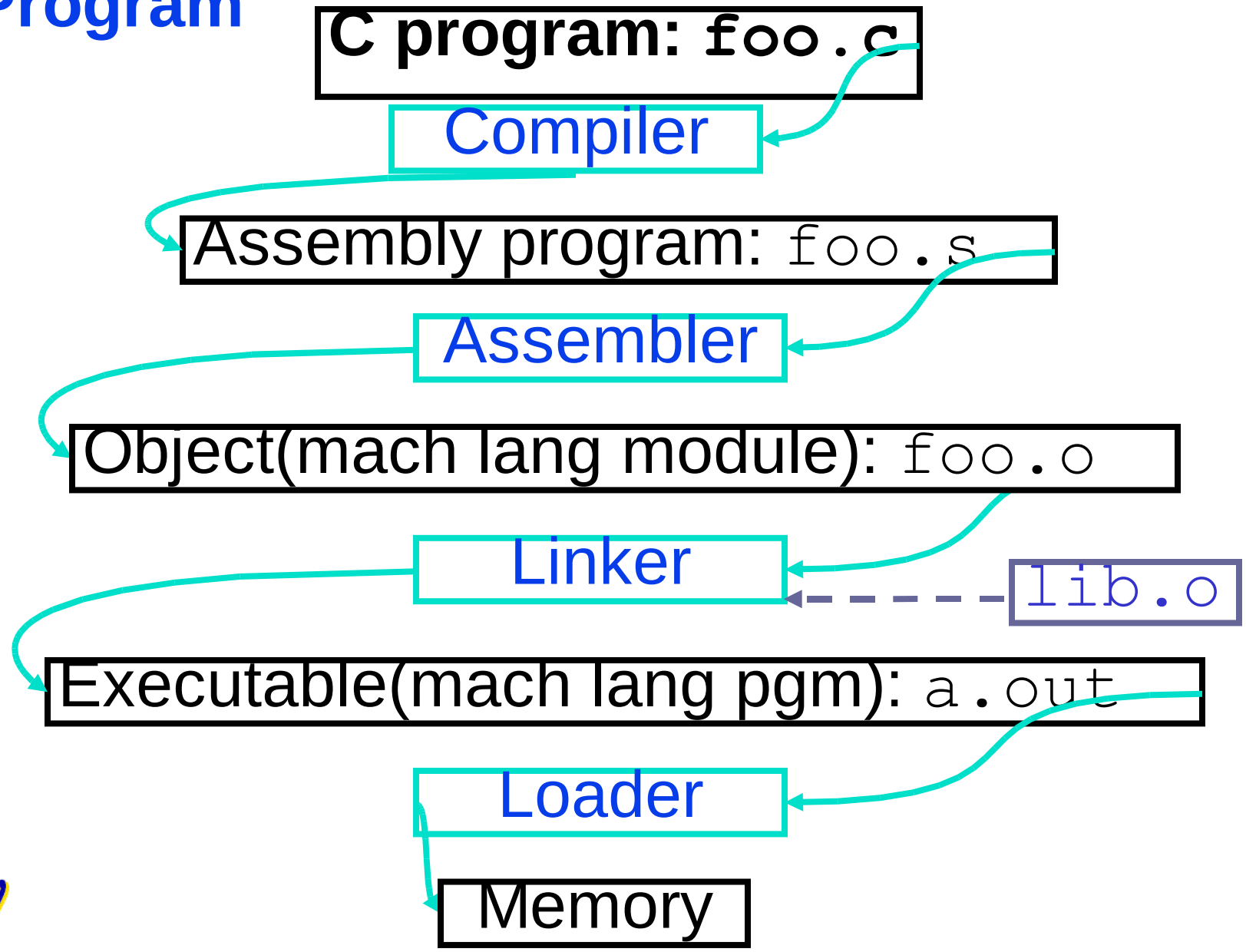
  else:

  ```
  lui reg,upper 16 bits of value
  ori reg,$zero,lower 16 bits
  ```

True Assembly Language

- **Problem:**
  - **When breaking up a pseudoinstruction, the assembler may need to use an extra reg.**
  - **If it uses any regular register, it'll overwrite whatever the program has put into it.**
- **Solution:**
  - **Reserve a register (`$1`, called `$at` for "assembler temporary") that assembler will use to break up pseudo-instructions.**
  - **Since the assembler may use this at any time, it's not safe to code with it.**

Steps to Starting a Program



Compiler

- Input: High-Level Language Code (e.g., C, Java such as `foo.c`)
- Output: Assembly Language Code (e.g., `foo.s` for MIPS)
- Note: Output *may* contain pseudoinstructions
- Pseudoinstructions: instructions that assembler understands but not in machine (last lecture) For example:
- `mov $s1,$s2` ⇒ `or $s1,$s2,$zero`

Assembler

- Input: MAL Assembly Language Code (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (e.g., `foo.o` for MIPS)
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

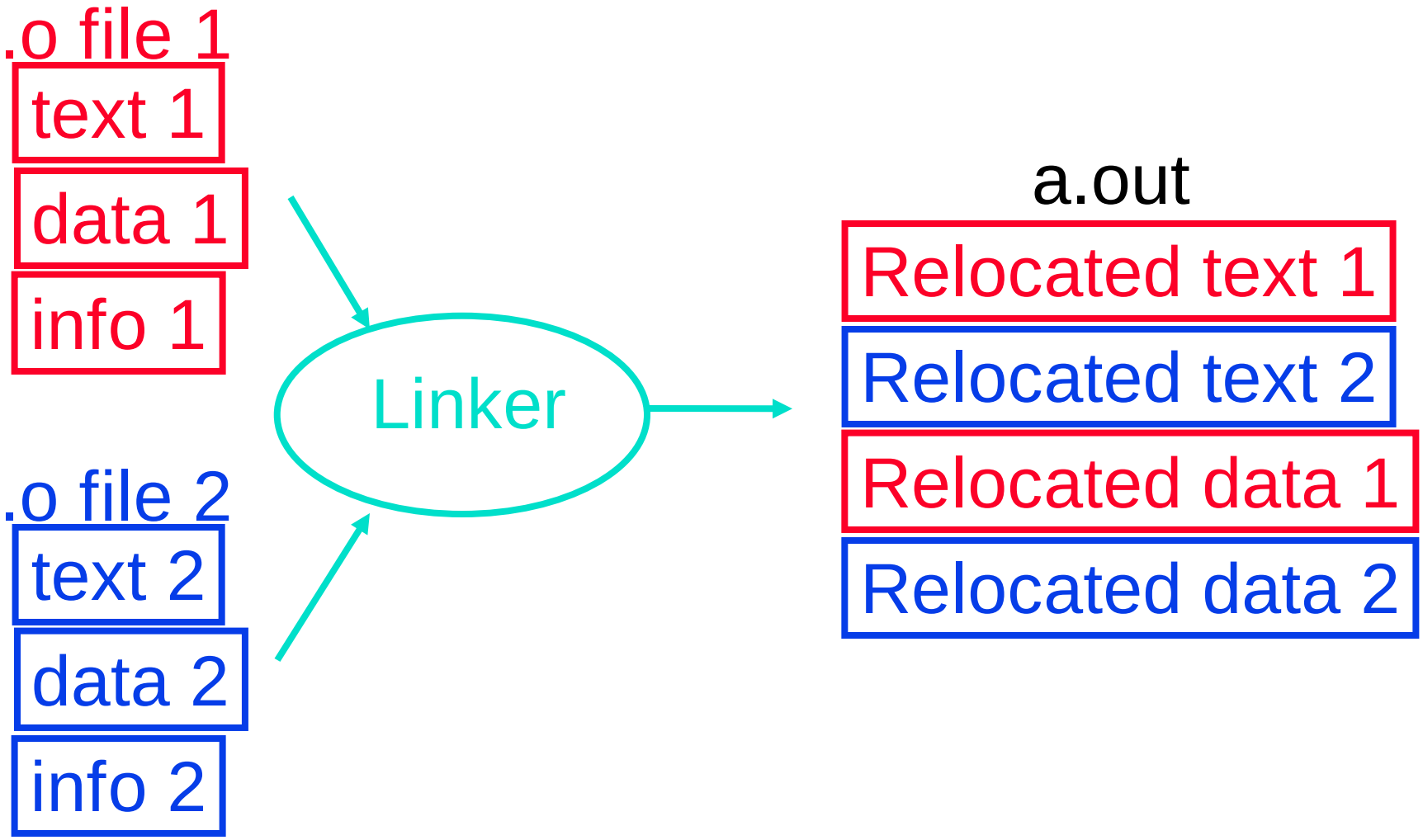
Producing Machine Language

- Constraint on Assembler:
  - The object file output (foo.o) may be only one of many object files in the final executable:
    - C: #include “my_helpers.h”
    - C: #include <stdio.h>
- Consequences:
  - Object files won’t know their base addresses until they are linked/loaded!
  - References to addresses will have to be adjusted in later stages

Object File Format

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation information: identifies lines of code that need to be “handled”
- symbol table: list of this file’s labels and data that can be referenced
- debugging information

Link Editor/Linker



Link Editor/Linker

- Step 1: Take text segment from each .o file and put them together.
- Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- Step 3: Resolve References
  - Go through Relocation Table and handle each entry
  - That is, fill in all absolute addresses

Loader

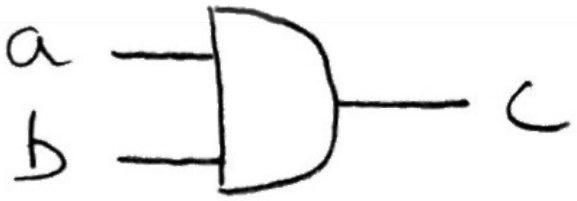
- Input: Executable Code (e.g., `a.out` for MIPS)
- Output: (program is run)
- Executable files are stored on disk.
- When one is run, loader's job is to load it into memory and start it running.
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks

True Assembly Language

- **MAL (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions**
- **TAL (True Assembly Language): set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)**
- **A program must be converted from MAL into TAL before translation into 1s & 0s.**

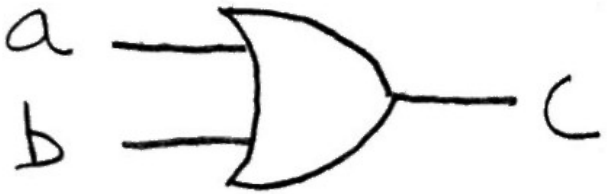
Logic Gates

AND



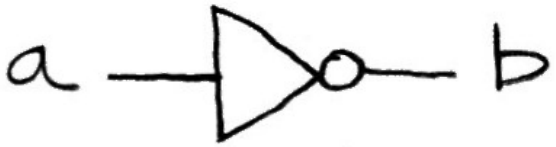
ab	c
00	0
01	0
10	0
11	1

OR



ab	c
00	0
01	1
10	1
11	1

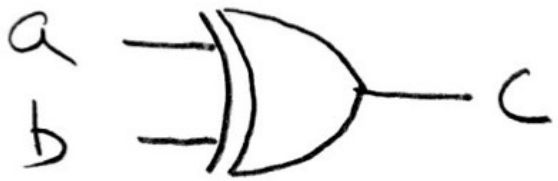
NOT



a	b
0	1
1	0

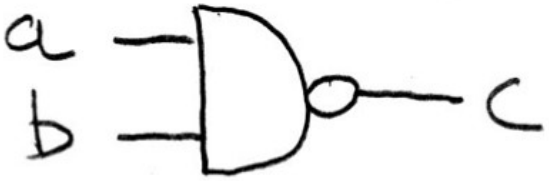
Logic Gates

XOR



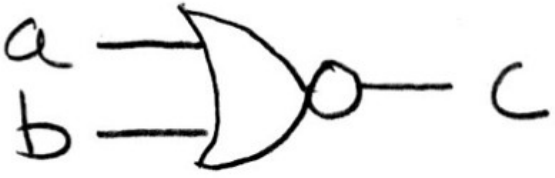
ab	c
00	0
01	1
10	1
11	0

NAND



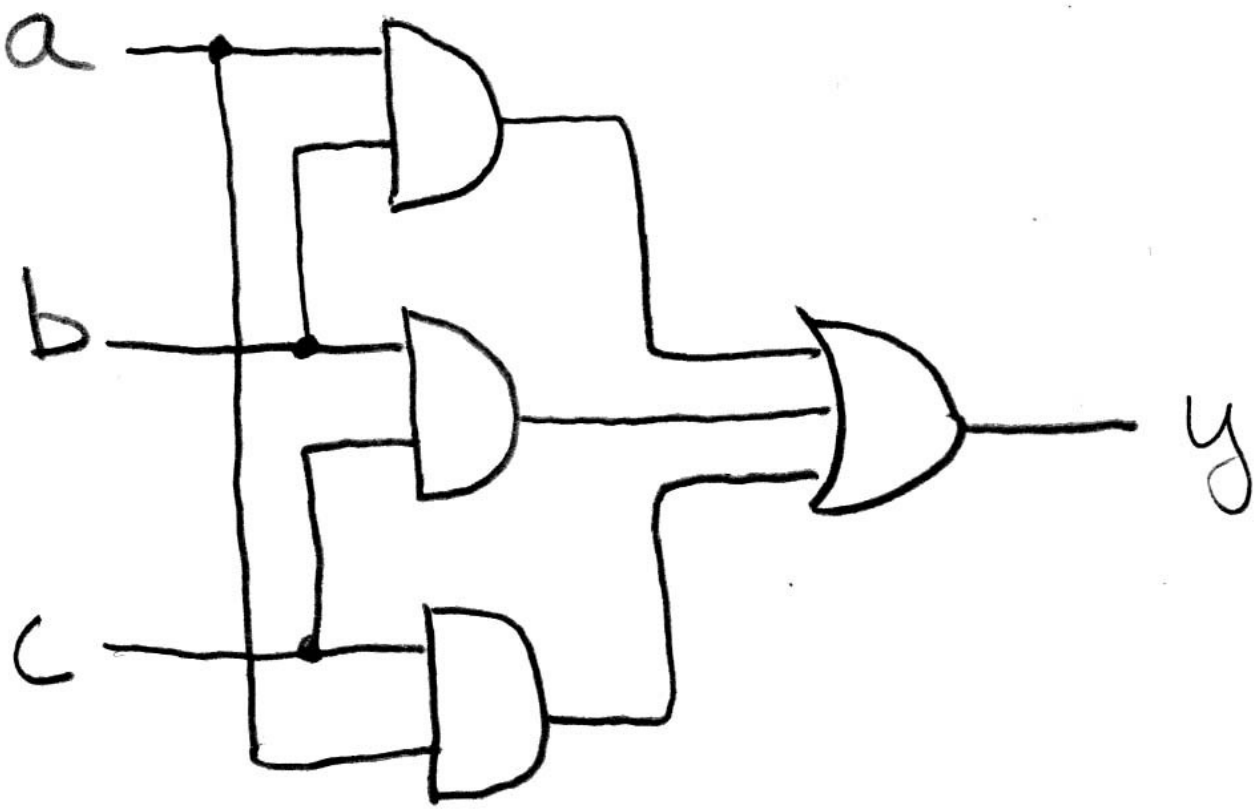
ab	c
00	1
01	1
10	1
11	0

NOR



ab	c
00	1
01	0
10	0
11	0

Boolean Algebra



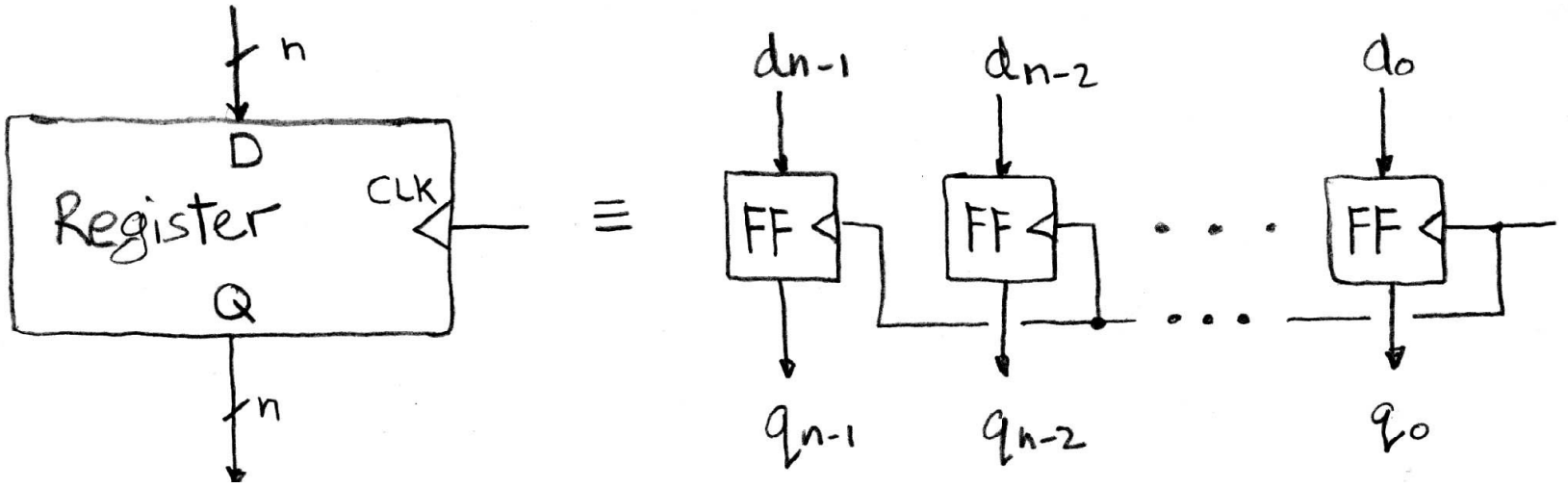
$$y = a \cdot b + a \cdot c + b \cdot c$$

$$y = ab + ac + bc$$

Laws of Boolean Algebra

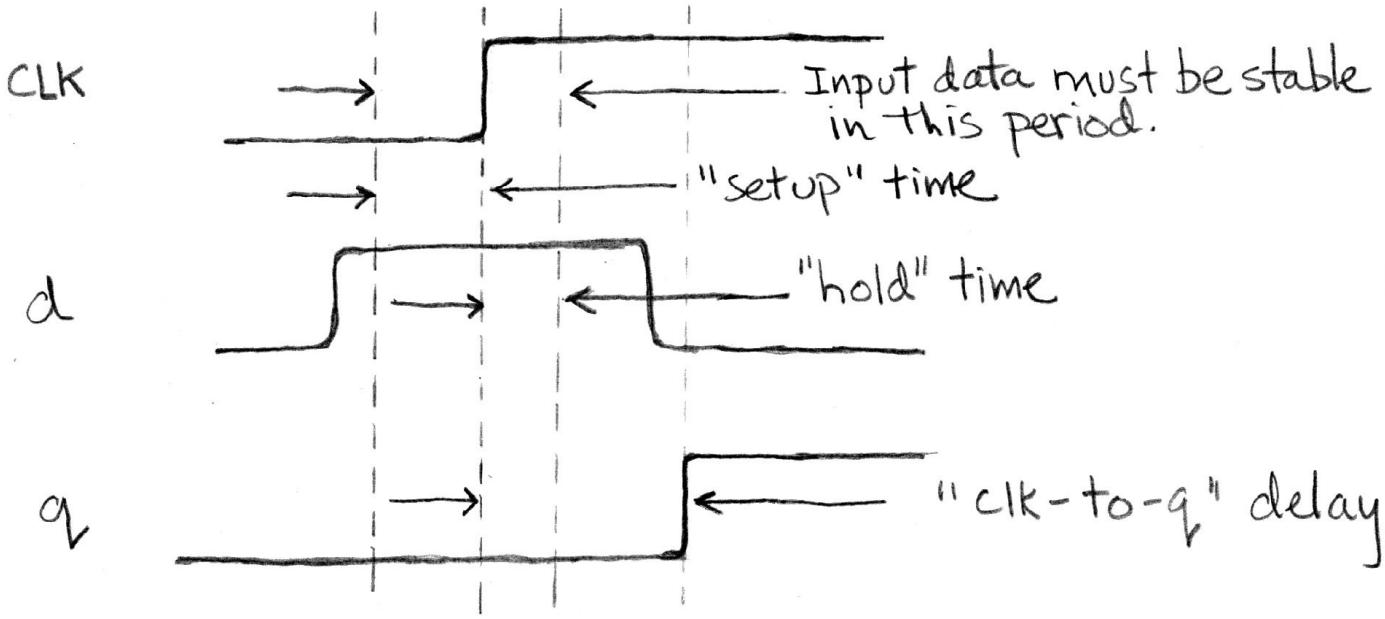
$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$	complementarity
$x \cdot 0 = 0$	$x + 1 = 1$	laws of 0's and 1's
$x \cdot 1 = x$	$x + 0 = x$	identities
$x \cdot x = x$	$x + x = x$	idempotent law
$x \cdot y = y \cdot x$	$x + y = y + x$	commutativity
$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$	associativity
$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$	distribution
$xy + x = x$	$(x + y)x = x$	uniting theorem
$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{(x + y)} = \overline{x} \cdot \overline{y}$	DeMorgan's Law

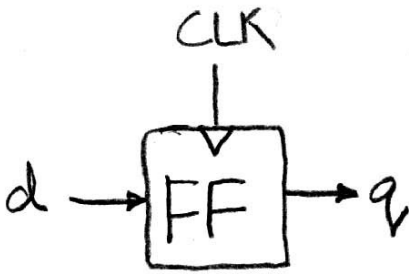
Register Details…What's in it anyway?



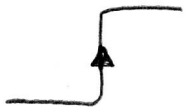
- n instances of a "Flip-Flop", called that because the output flips and flops betw. 0,1
- D is "data"
- Q is "output"
- Also called "d-q Flip-Flop","d-type Flip-Flop"

What's the timing of a Flip-flop?



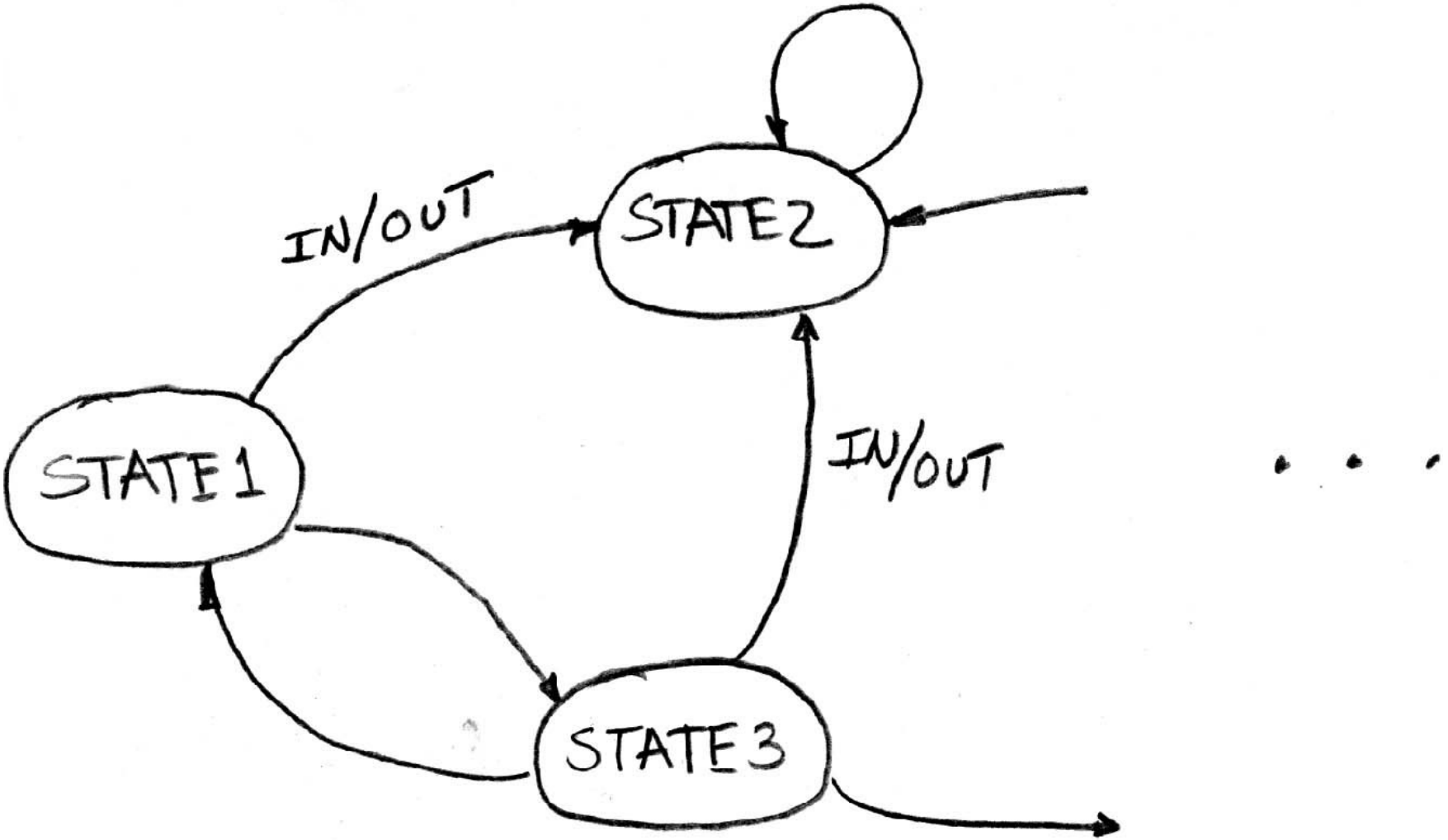


- **Edge-triggered D-type flip-flop**
  - **This one is "positive edge-triggered"**

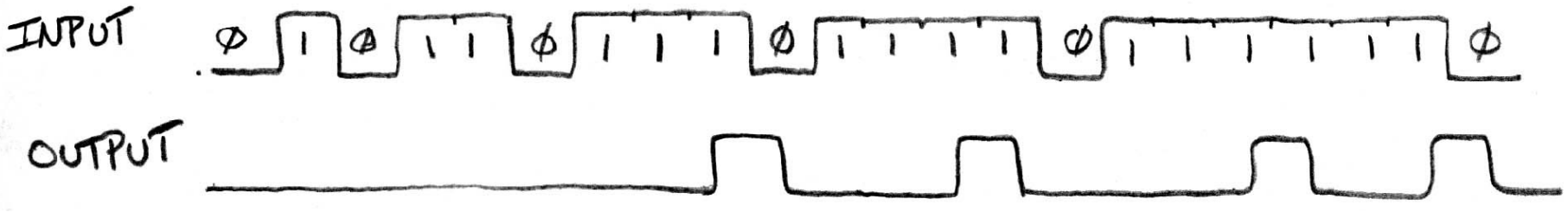


- **"On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored."**

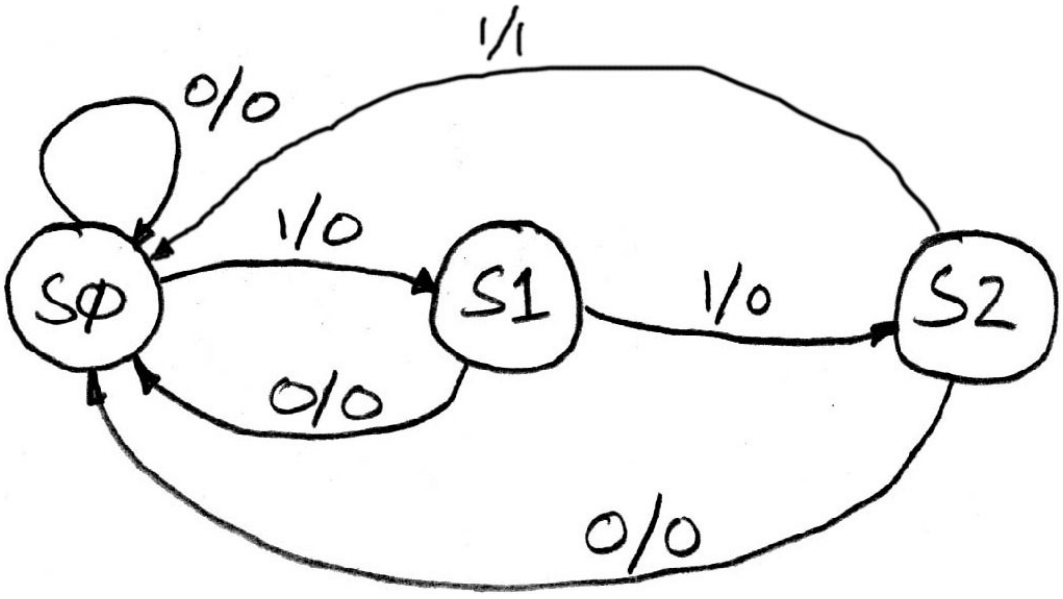
Finite State Machines Introduction



Finite State Machine Example: 3 ones…

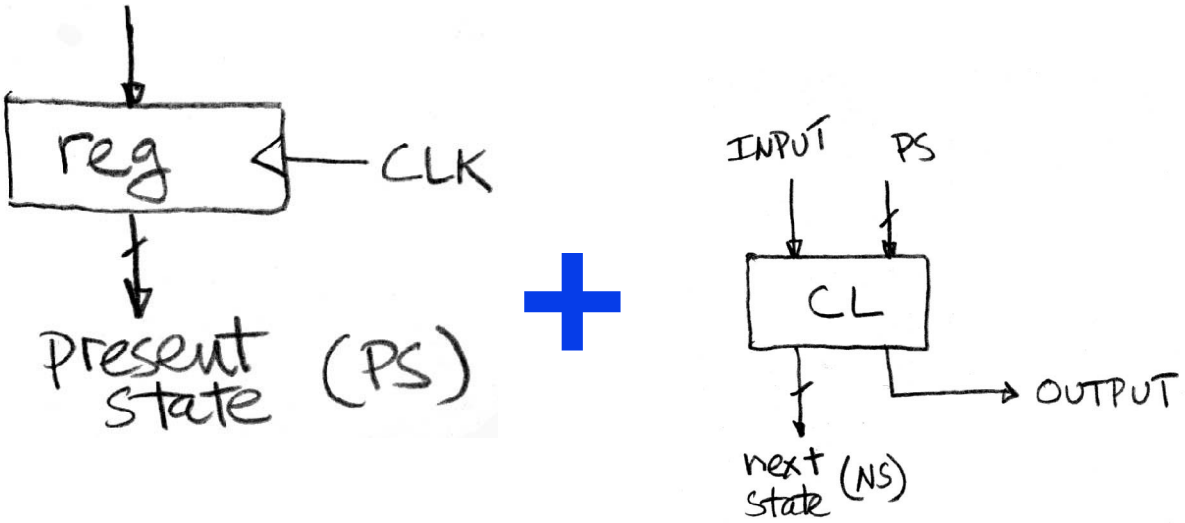


Draw the FSM…

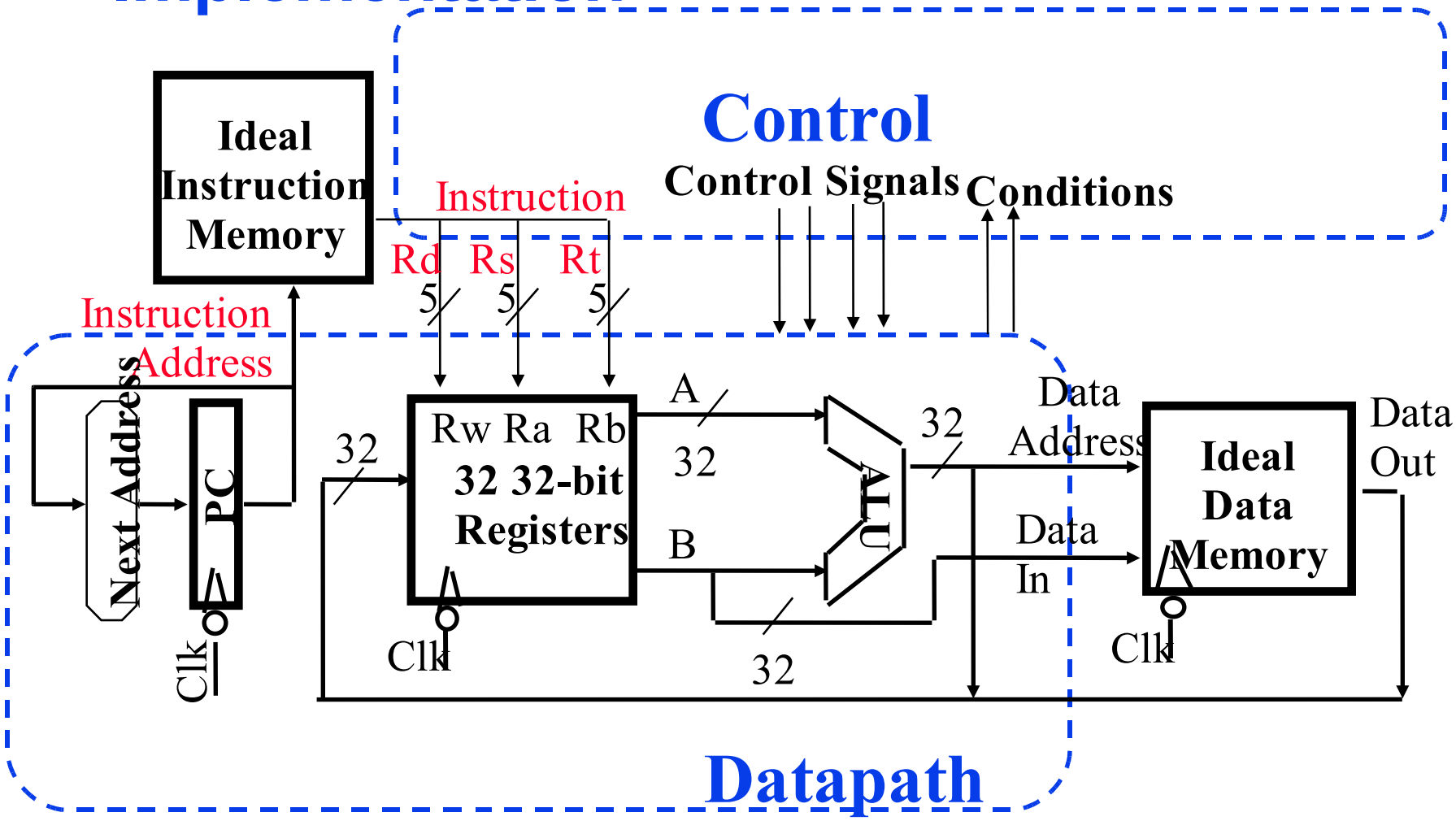


PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

Hardware Implementation of FSM

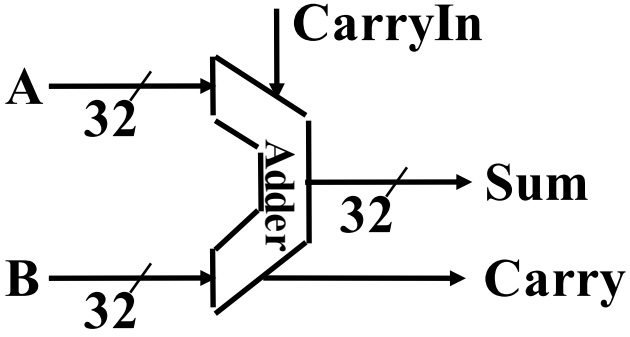


Step 1: Abstract Implementation

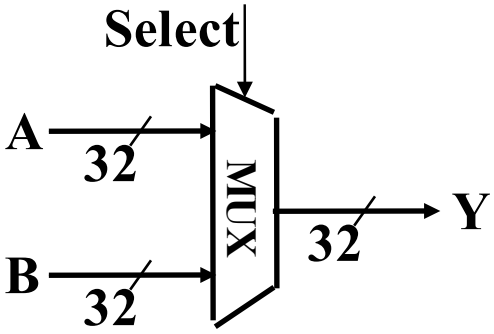


Combinational Logic: More Elements

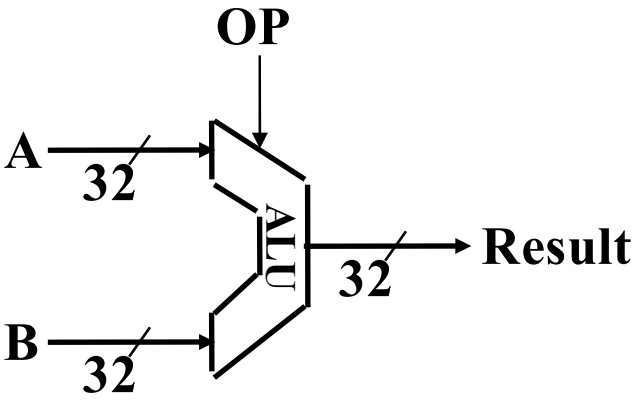
- Adder



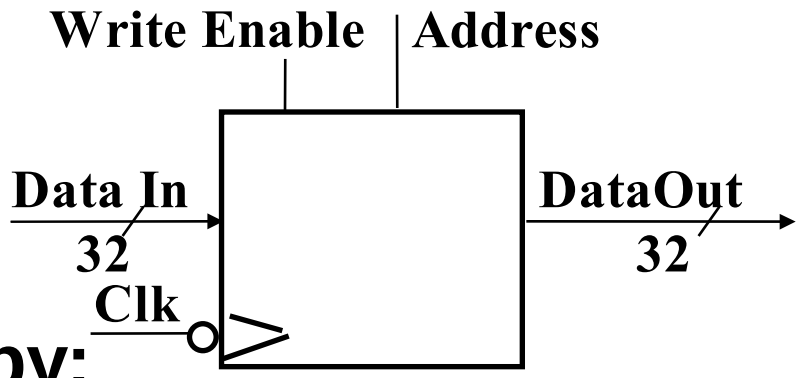
- MUX



- ALU



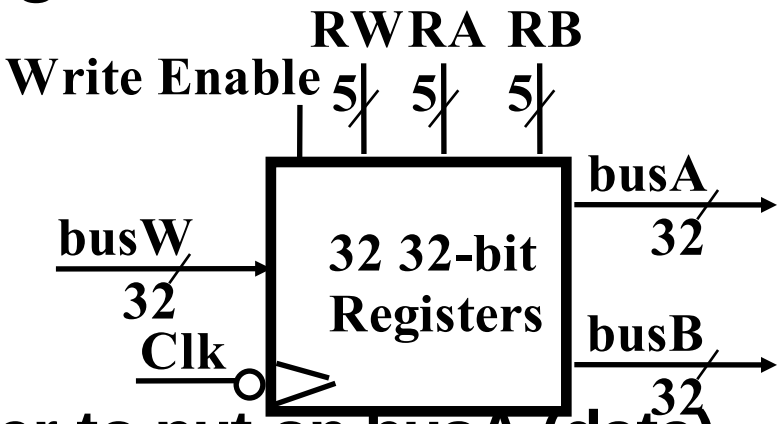
Storage Element: Idealized Memory



- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after "access time."

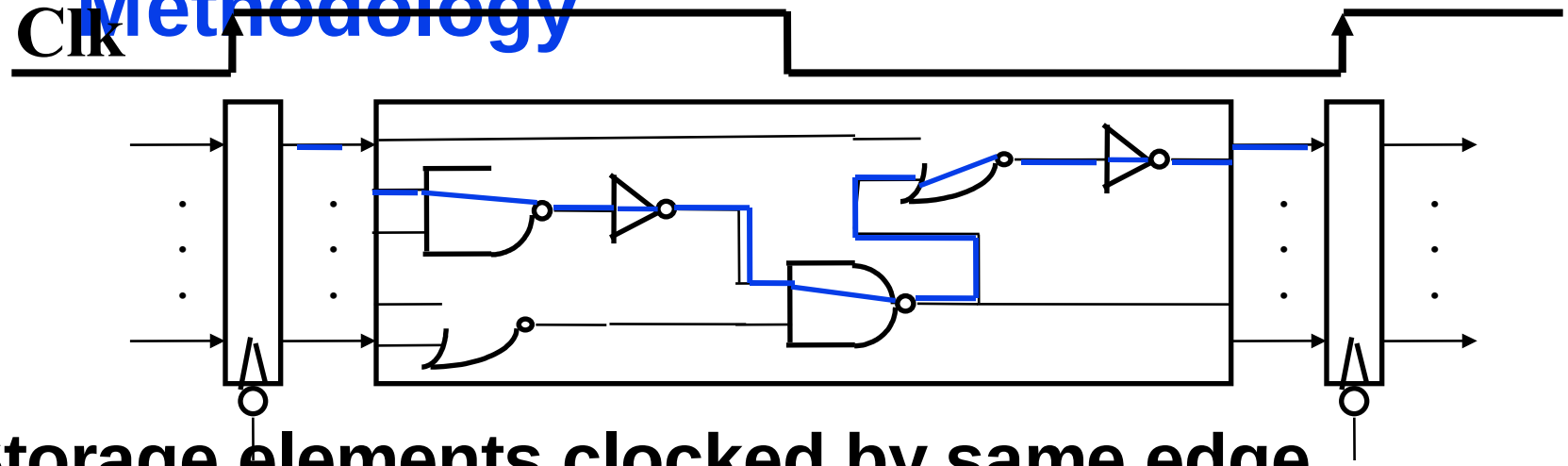
Storage Element: Register File

- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW



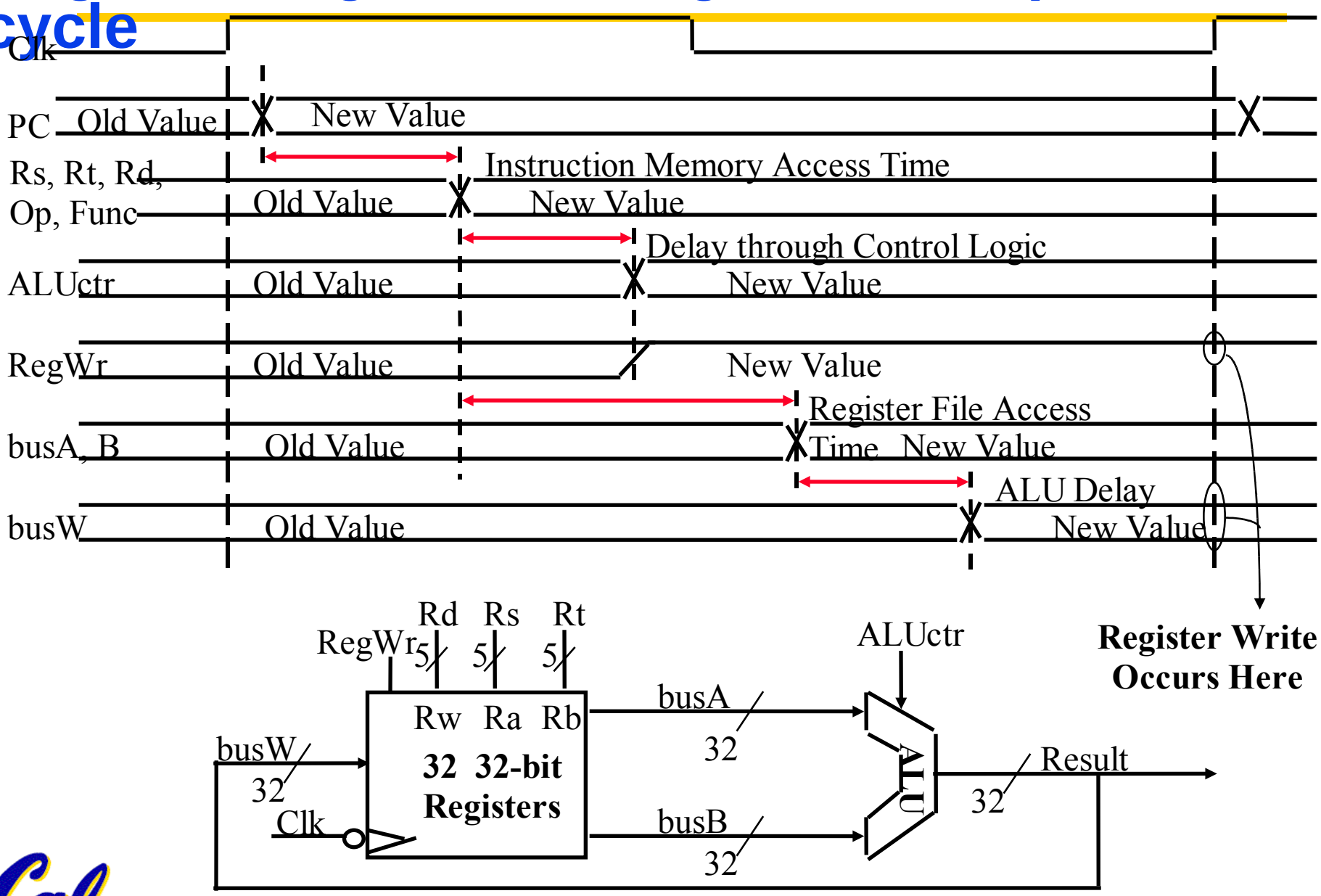
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after "access time."

Clocking Methodology

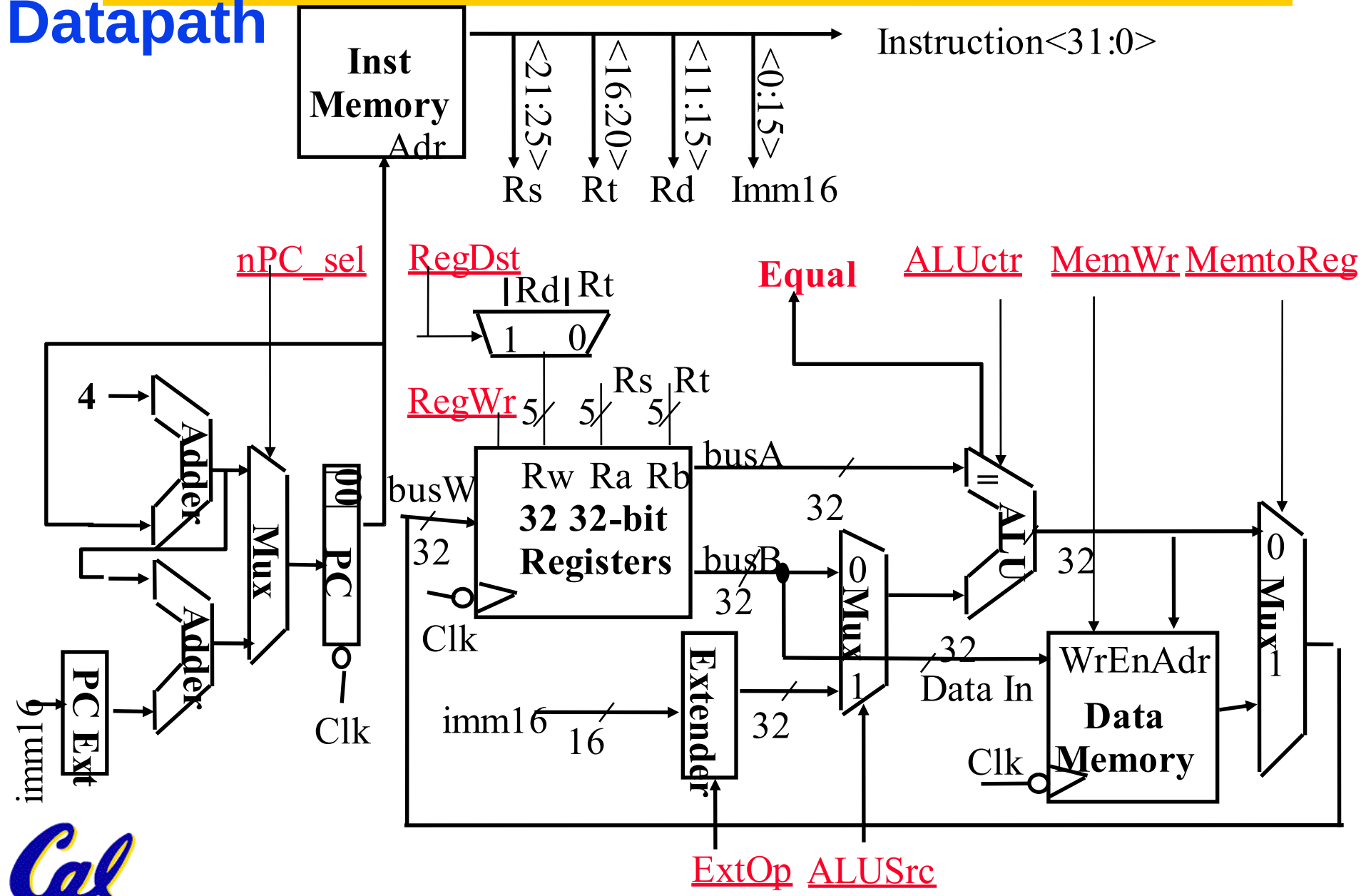


- Storage elements clocked by same edge
- Being physical devices, flip-flops (FF) and combinational logic have some delays
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF, and we have the usual clock-to-Q delay
- "Critical path" (longest path through logic) determines length of clock period

Register-Register Timing: One complete cycle



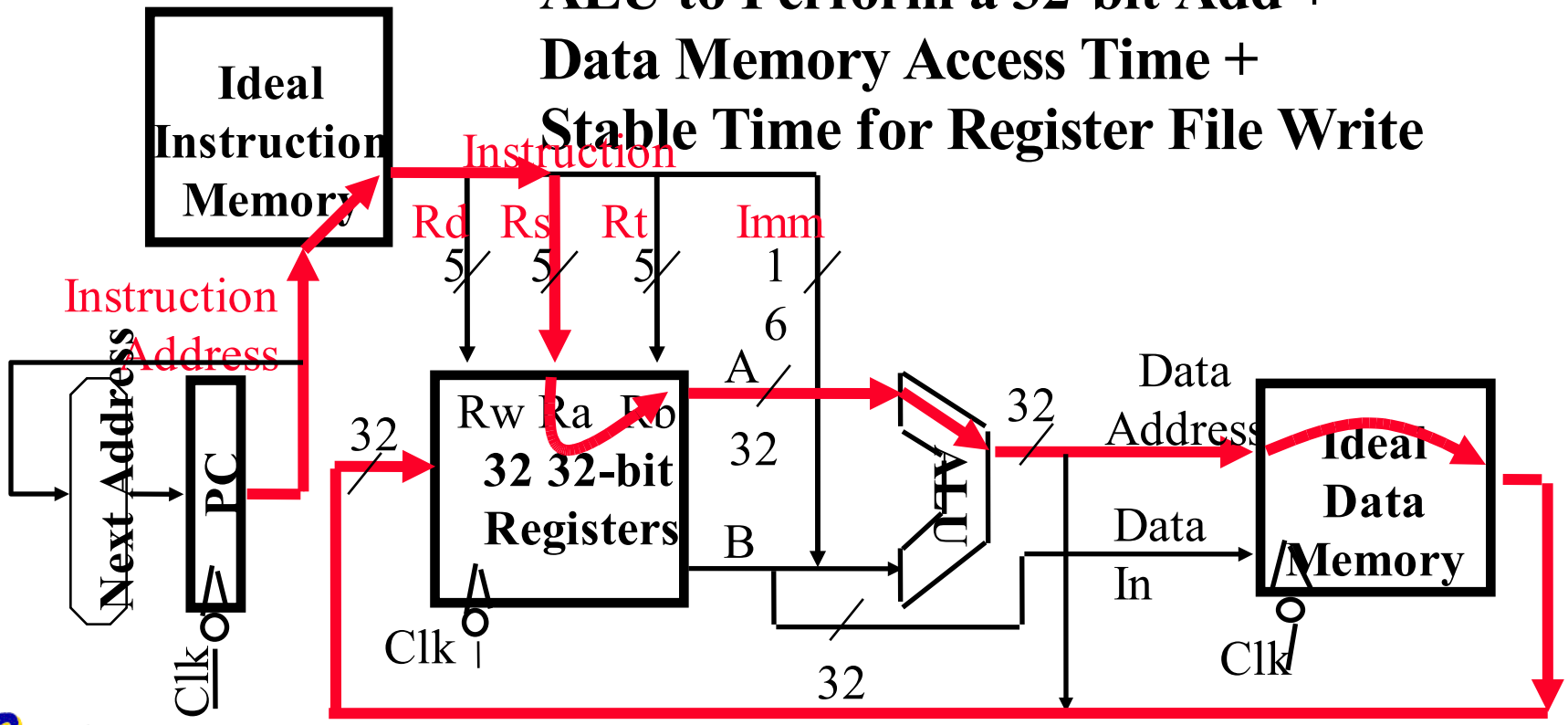
Putting it All Together:A Single Cycle Datapath



An Abstract View of the Critical Path

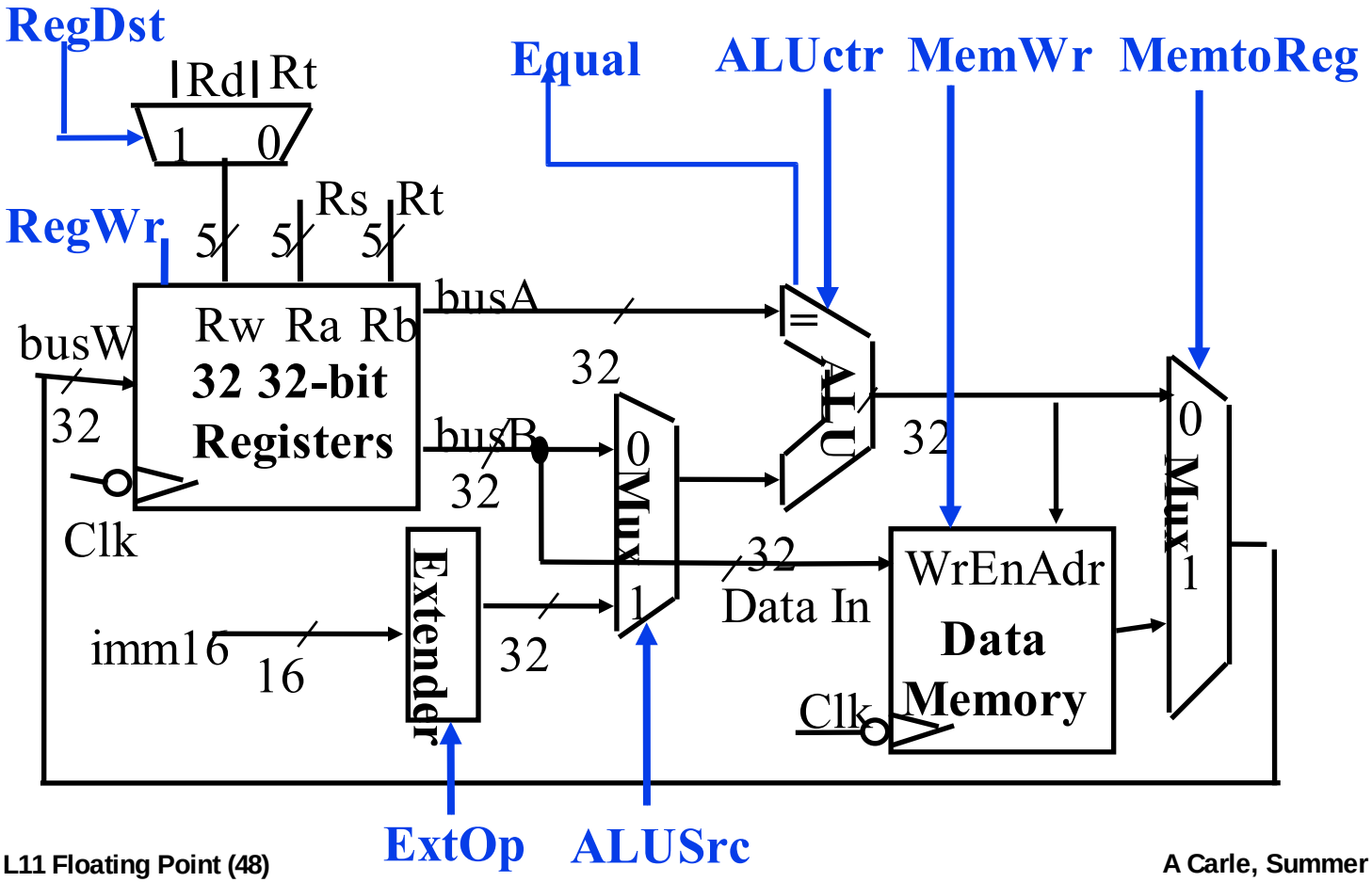
- This affects how fast you can clock your PC

Critical Path (Load Operation) =
Delay clock through PC (FFs) +
Instruction Memory's Access Time +
Register File's Access Time, +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Stable Time for Register File Write



Recap: Meaning of the Control Signals

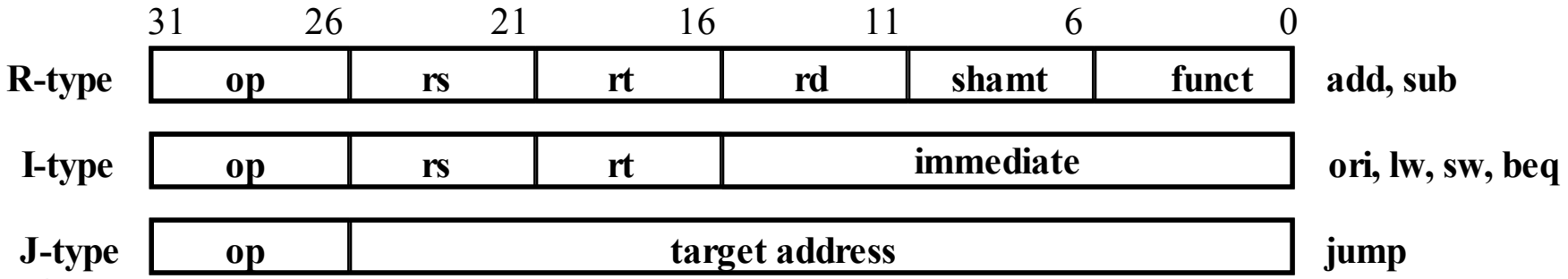
- ExtOp: "zero", "sign"
- ALUsrc: 0 ⇒ regB; 1 ⇒ immed
- ALUctr: "add", "sub", "or"
- MemWr: 1 ⇒ write memory
- MemtoReg: 0 ⇒ ALU; 1 ⇒ Mem
- RegDst: 0 ⇒ "rt"; 1 ⇒ "rd"
- RegWr: 1 ⇒ write register



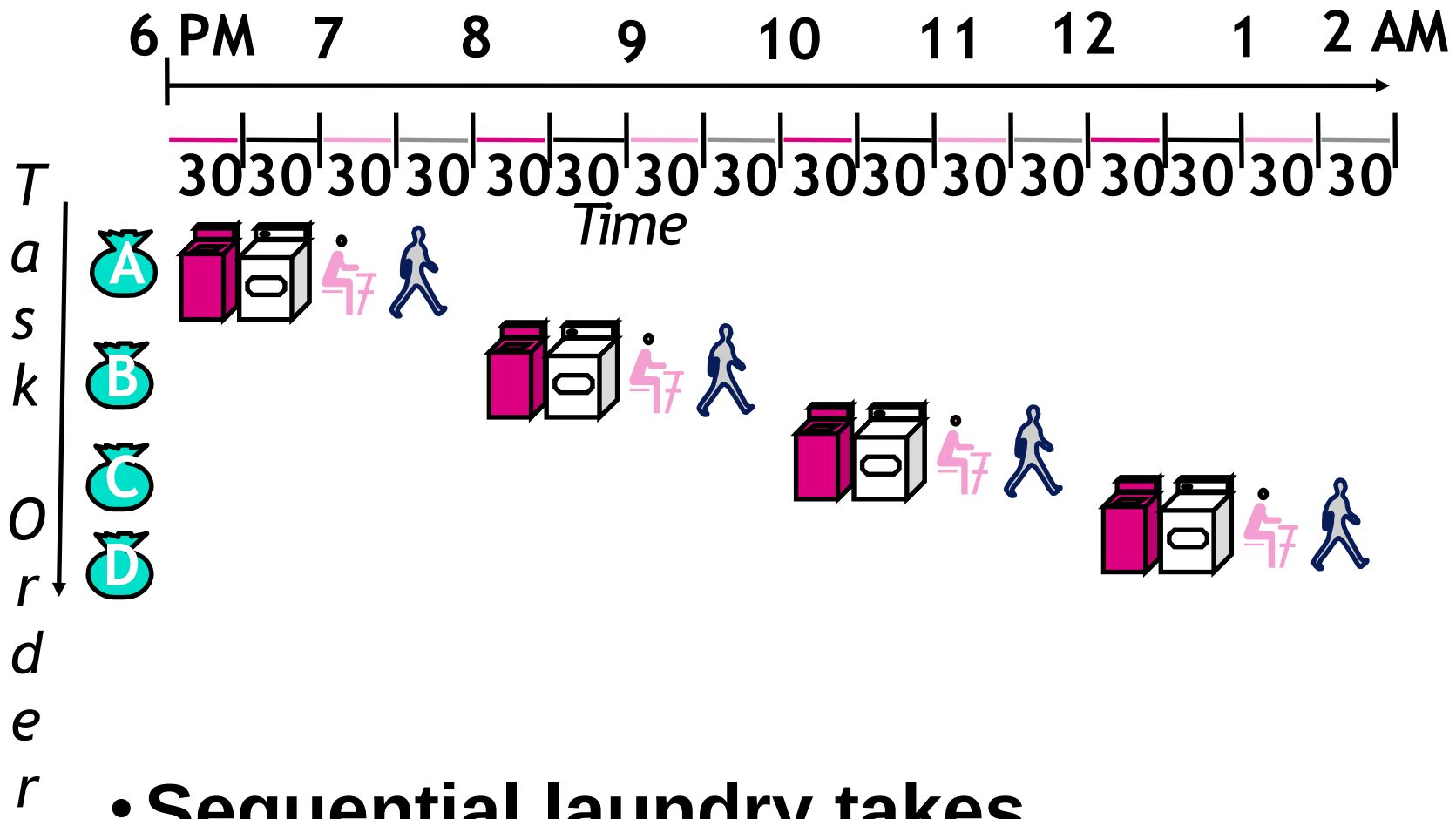
A Summary of the Control Signals

See Appendix A

	add	sub	ori	lw	sw	beq	jump
func	10 0000	10 0010	**We Don't Care :-)**				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
nPCsel	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx



Sequential Laundry

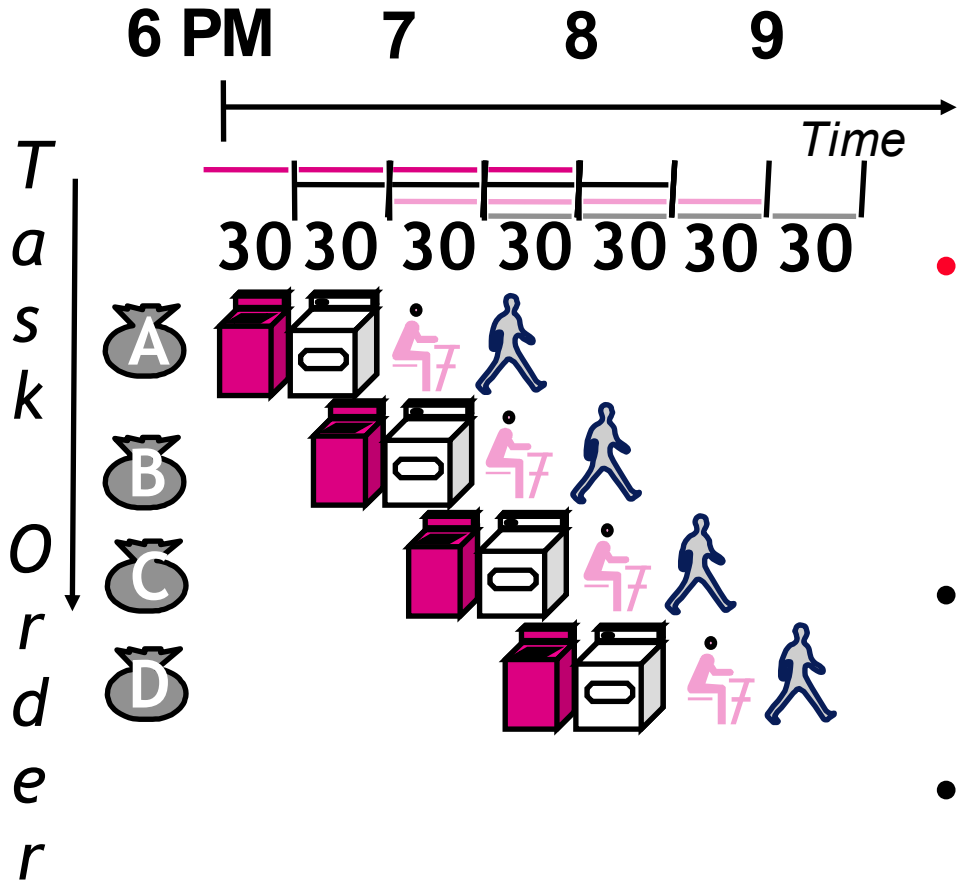


- Sequential laundry takes 8 hours for 4 loads

General Definitions

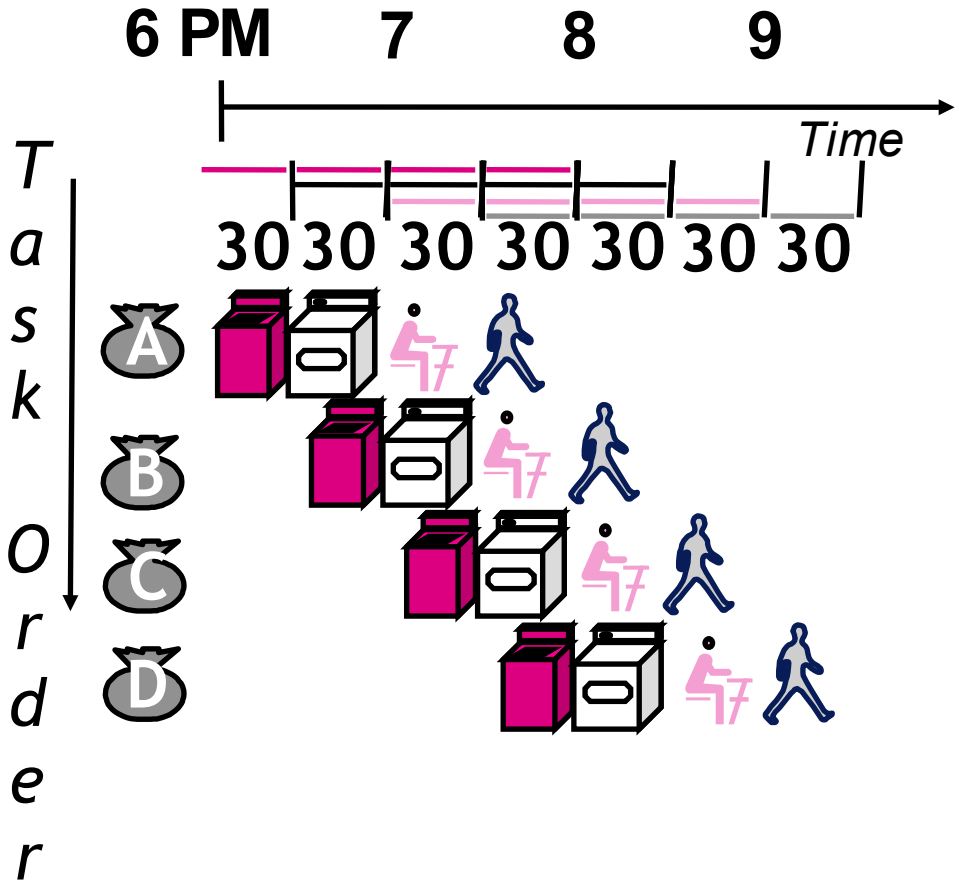
- **Latency: time to completely execute a certain task**
  - **for example, time to read a sector from disk is disk access time or disk latency**
  - **Instruction latency is time from when instruction starts to time when it finishes.**
- **Throughput: amount of work that can be done over a period of time**

Pipelining Lessons (1/2)



- 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
- Time to "fill" pipeline and time to "drain" it reduces speedup: 2.3X v. 4X in this example

Pipelining Lessons (2/2)



- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages also reduces speedup

Steps in Executing MIPS

1) IFetch: Fetch Instruction, Increment PC

2) Decode Instruction, Read Registers

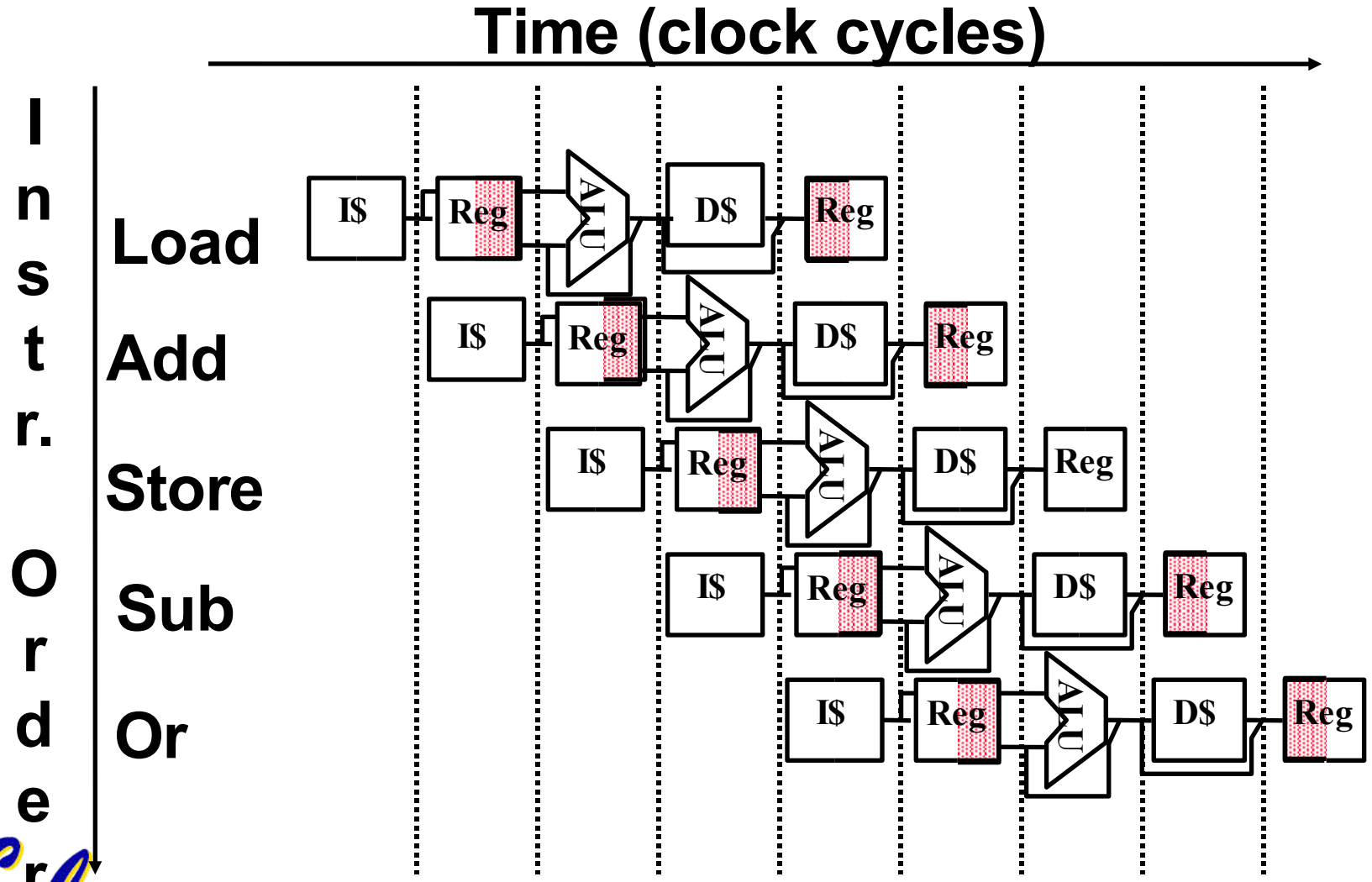
3) Execute:
   Mem-ref: Calculate Address
   Arith-log: Perform Operation

4) Memory:
   Load: Read Data from Memory
   Store: Write Data to Memory

5) Write Back: Write Data to Register

Graphical Pipeline Representation

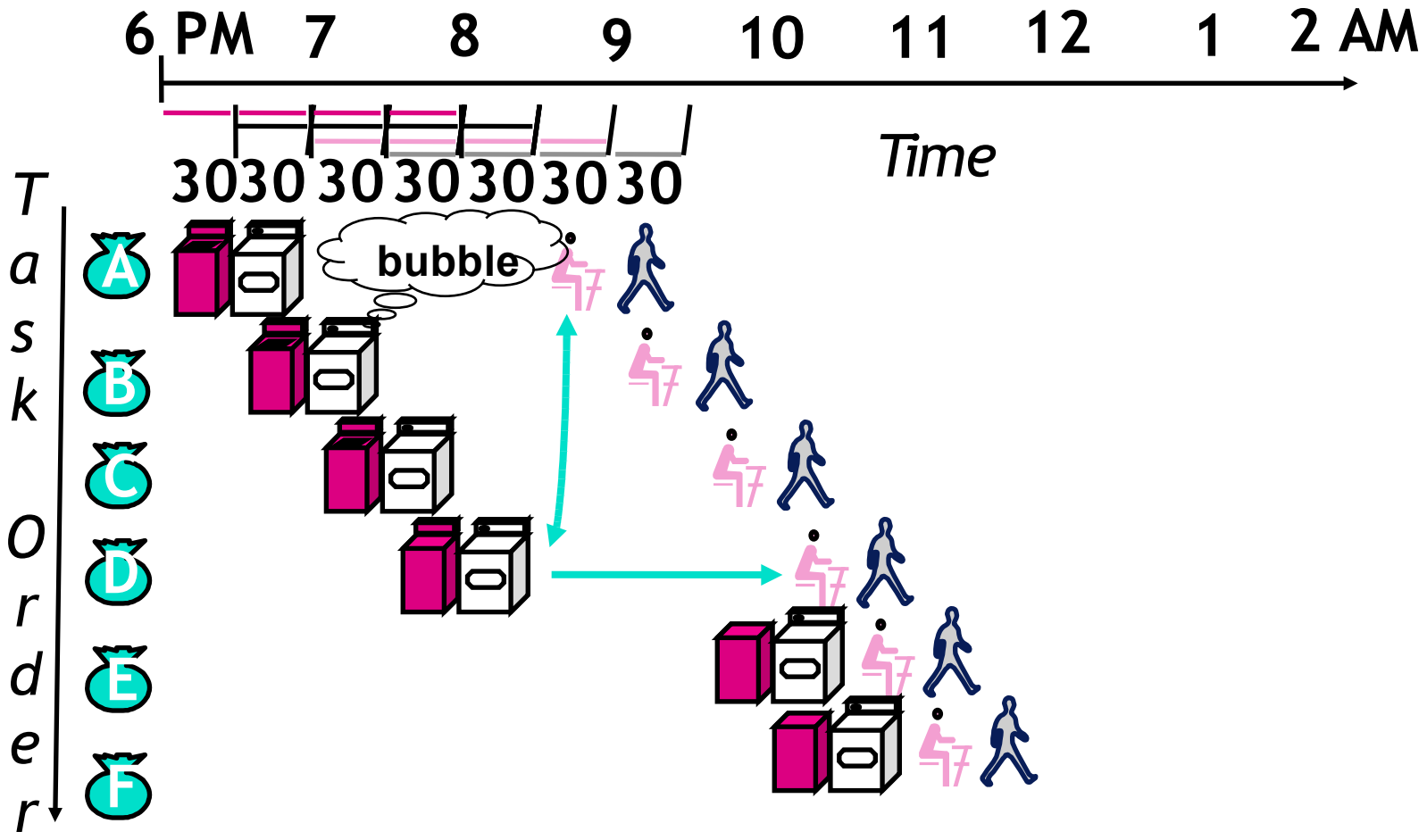
(In Reg, right half highlight read, left half write)



Things to Remember

- Optimal Pipeline
  - Each stage is executing part of an instruction each clock cycle.
  - One instruction finishes during each clock cycle.
  - *On average*, executes far more quickly.
- What makes this work?
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.

Pipeline Hazard: Matching socks in later load

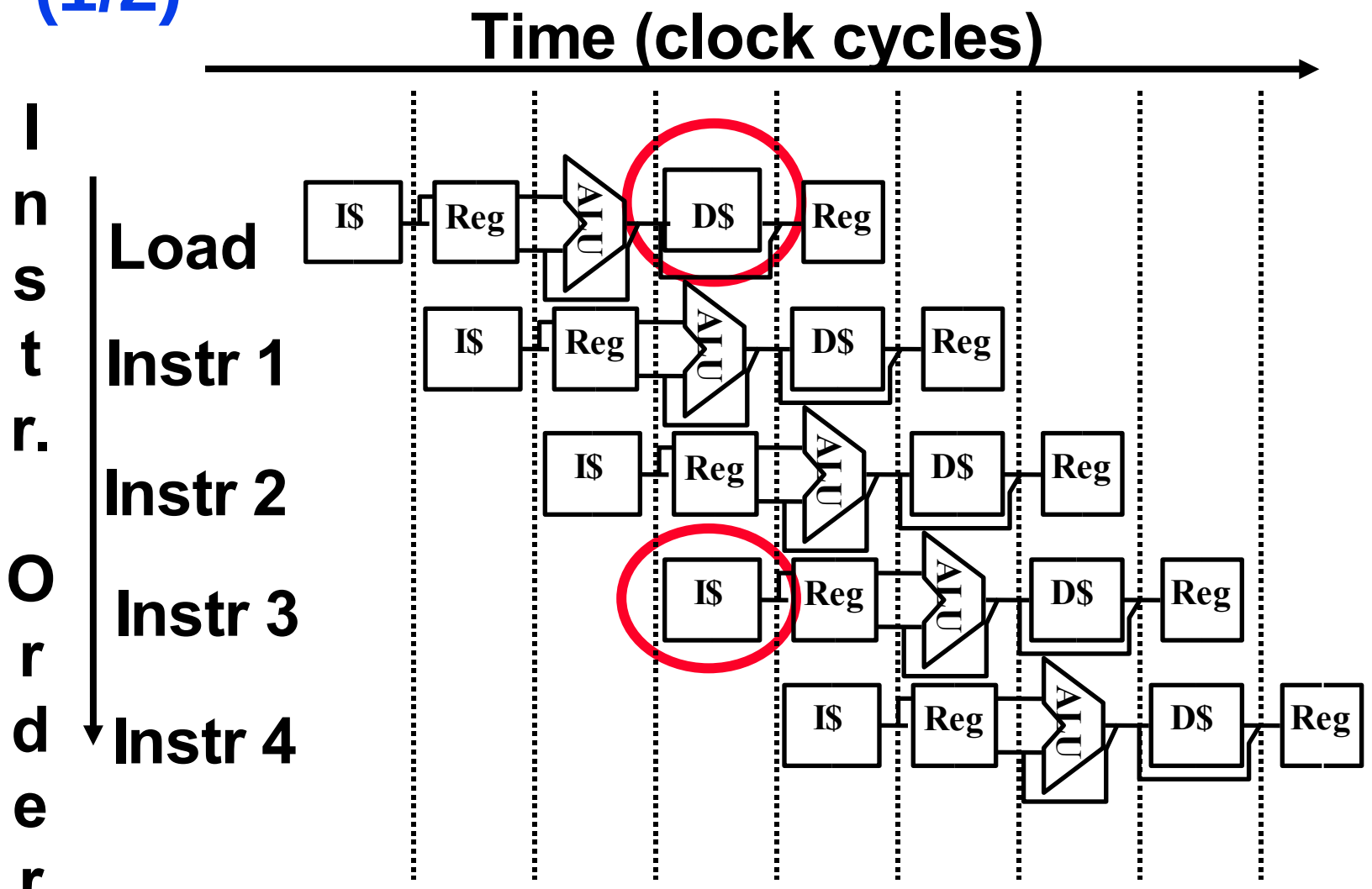


A depends on D; stall since folder tied up

Problems for Computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Control hazards: Pipelining of branches & other instructions stall the pipeline until the hazard; "bubbles" in the pipeline
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)

Structural Hazard #1: Single Memory (1/2)

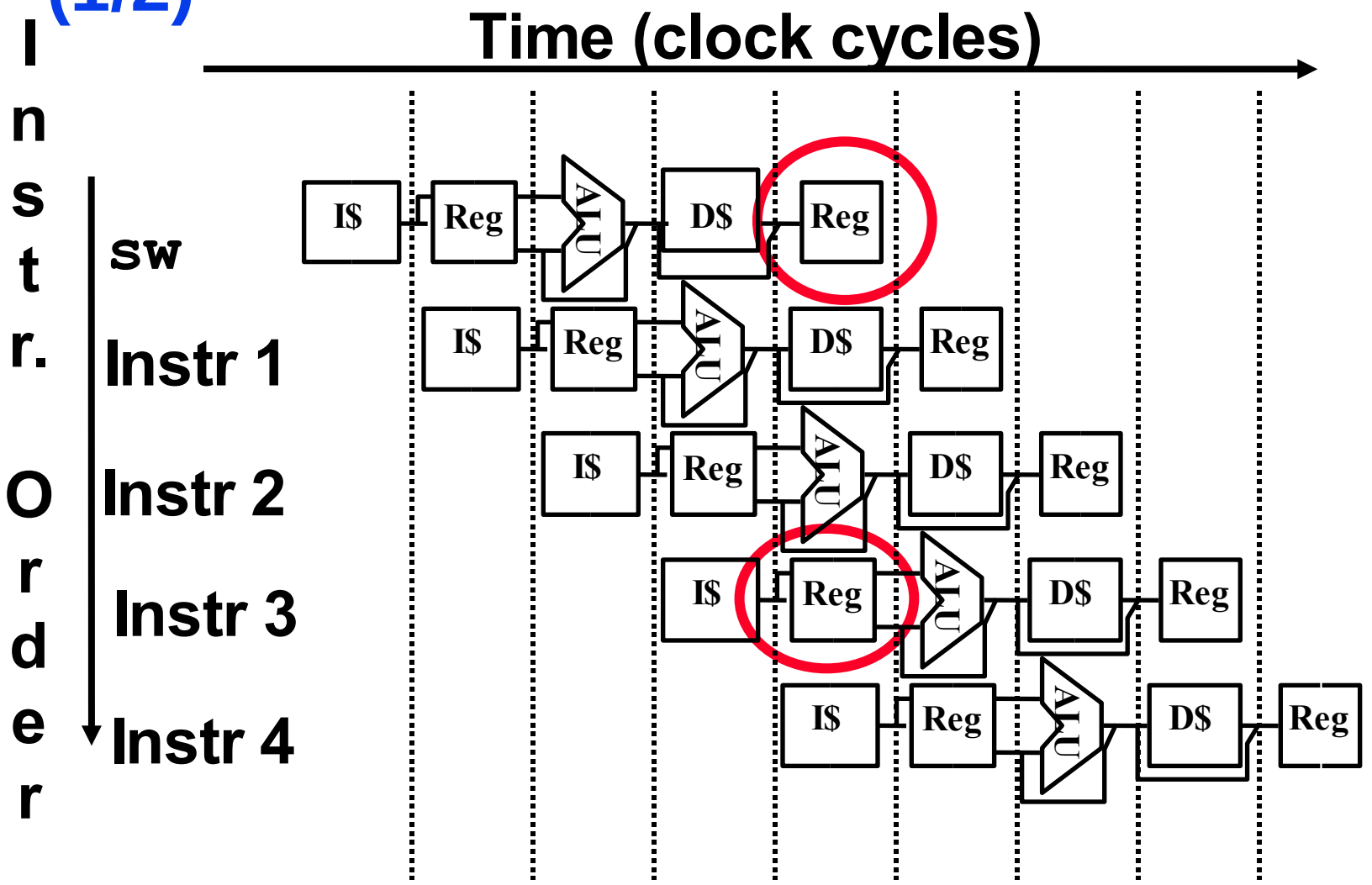


Read same memory twice in same clock cycle

Structural Hazard #1: Single Memory (2/2)

- Solution:
  - infeasible and inefficient to create second memory
  - (We'll learn about this more next week)
  - so simulate this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)
  - have both an L1 Instruction Cache and an L1 Data Cache
  - requires complex hardware to control when both caches miss!

Structural Hazard #2: Registers (1/2)

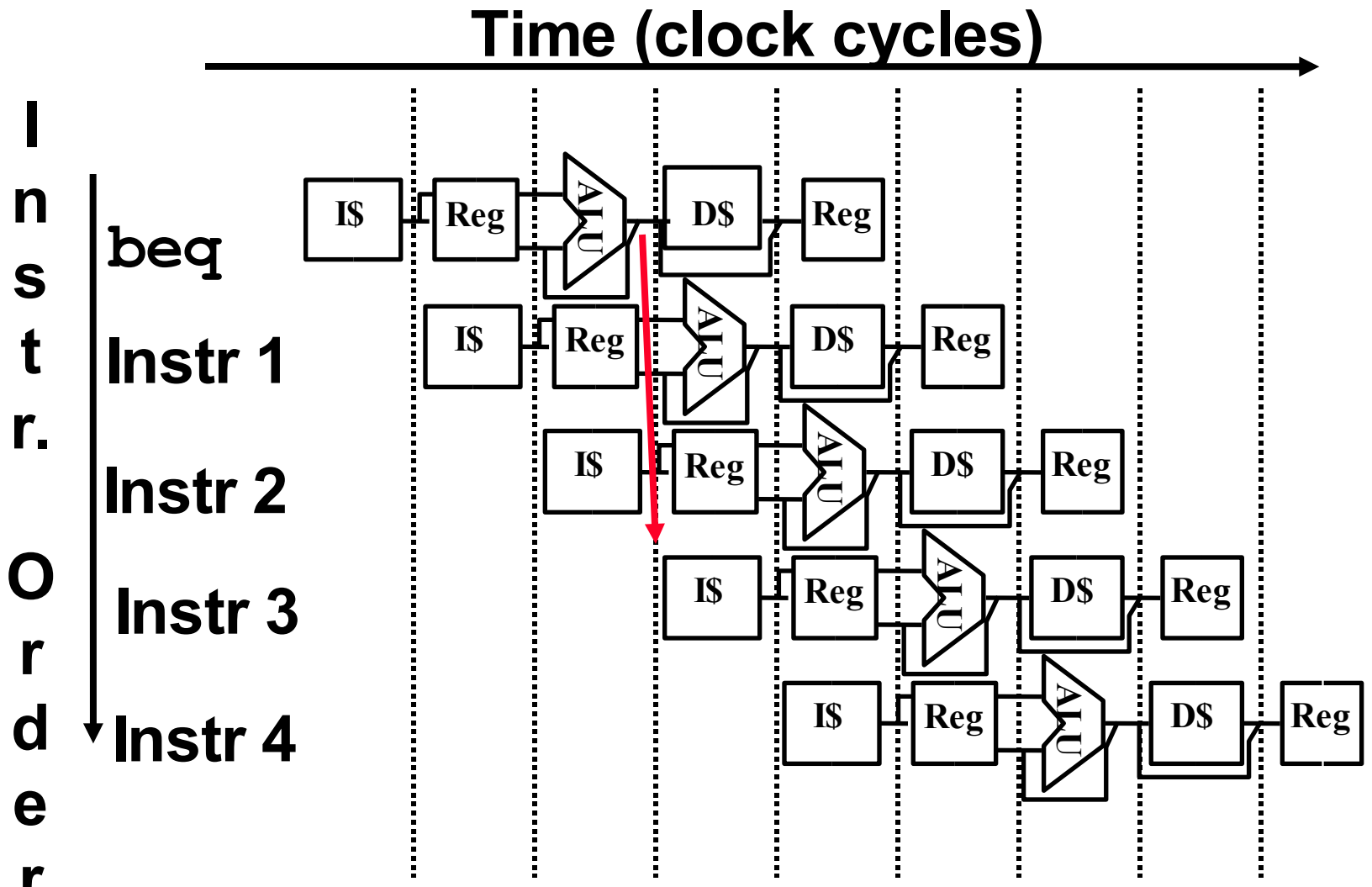


Can't read and write to registers simultaneously

Structural Hazard #2: Registers (2/2)

- Fact: Register access is *VERY* fast: takes less than half the time of ALU stage
- Solution: introduce convention
  - always Write to Registers during first half of each clock cycle
  - always Read from Registers during second half of each clock cycle (easy when async)
  - Result: can perform Read and Write during same clock cycle

Control Hazard: Branching



Where do we do the compare for the branch?

Control Hazard: Branching

- We put branch decision-making hardware in ALU stage
  - therefore two more instructions after the branch will *always* be fetched, whether or not the branch is taken
- Desired functionality of a branch
  - if we do not take the branch, don't waste any time and continue executing normally
  - if we take the branch, don't execute any instructions after the branch, just go to the desired label

Control Hazard: Branching

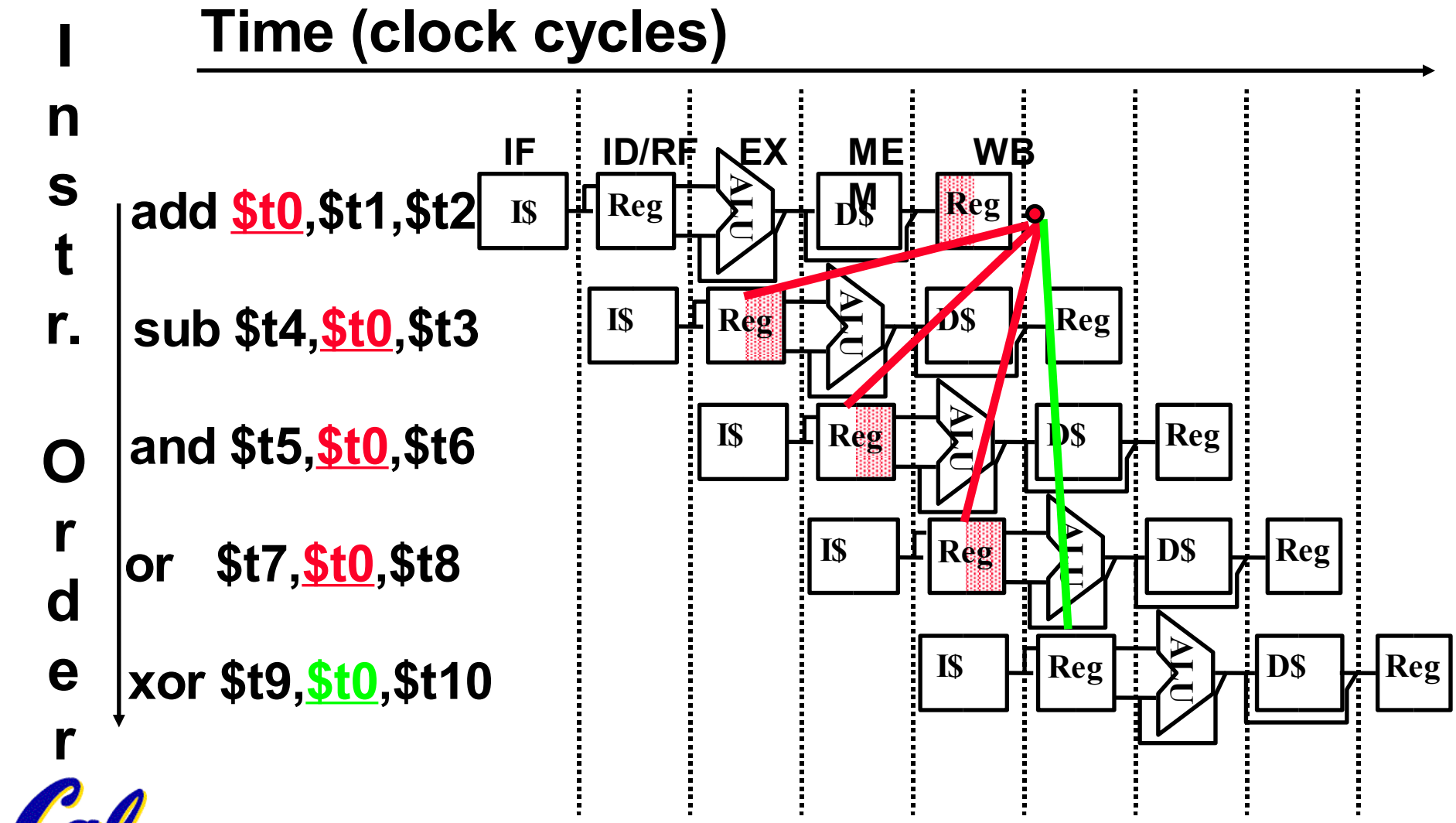
- Optimization #1:
  - move asynchronous comparator up to Stage 2
  - as soon as instruction is decoded (Opcode identifies is as a branch), immediately make a decision and set the value of the PC (if necessary)
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Side Note: This means that branches are idle in Stages 3, 4 and 5.

Control Hazard: Branching

- Optimization #2: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the branch-delay slot)

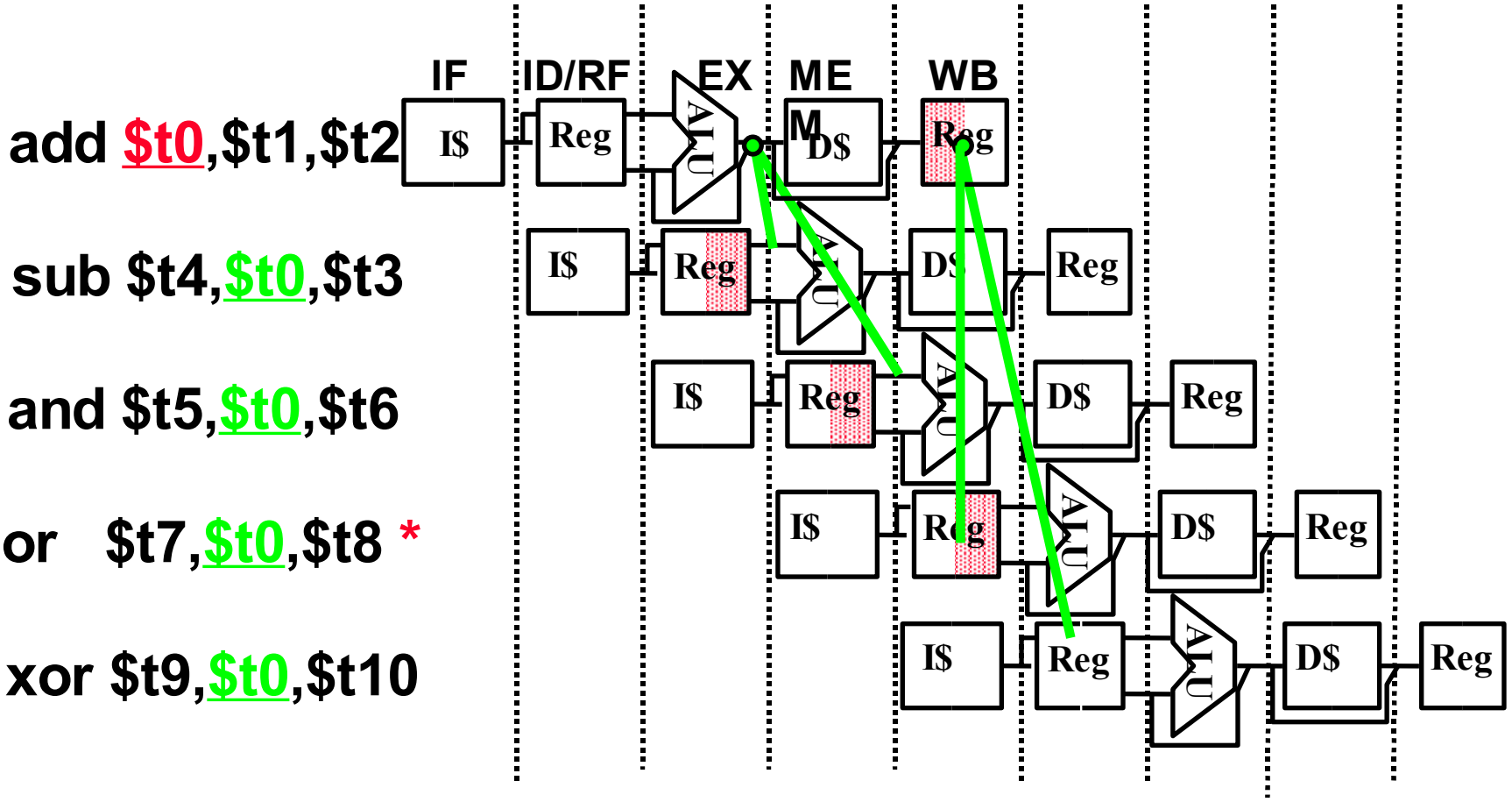
Data Hazards

$t0 not written back in time!



Data Hazard Solution: Forwarding

Fix by Forwarding result as soon as we have it to where we need it:



* "or" hazard solved by register hardware

Data Hazard: Loads

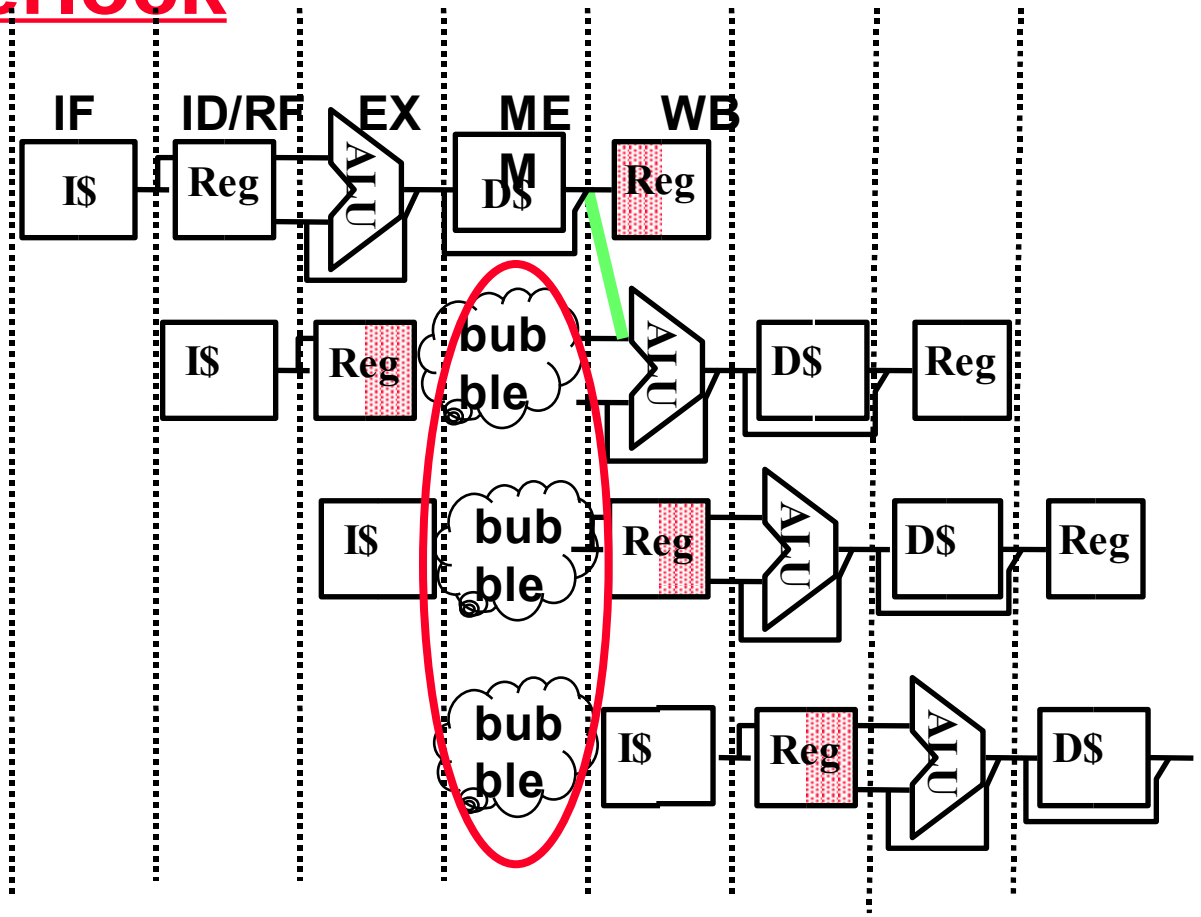
- Hardware must stall pipeline
- Called "interlock"

lw $t0, 0($t1)

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



Data Hazard: Loads

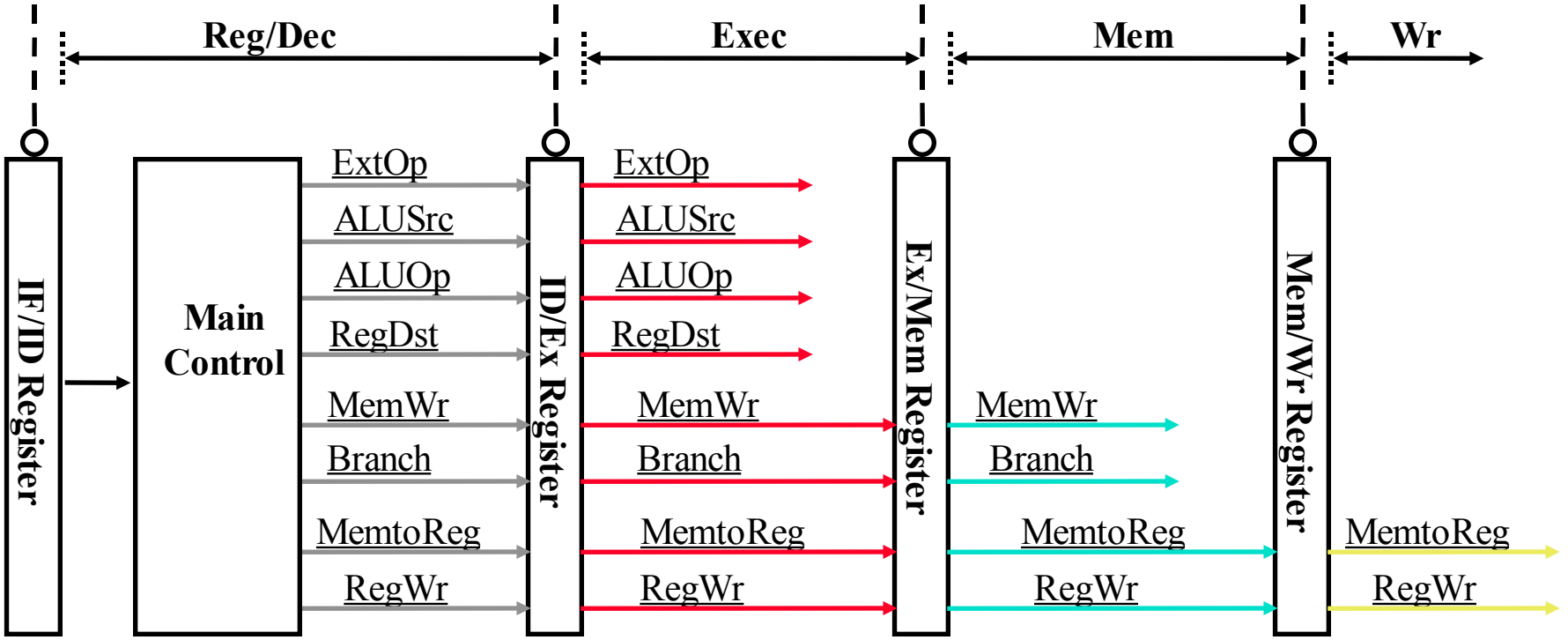
- Instruction slot after a load is called "load delay slot"
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)

C.f. Branch Delay vs. Load Delay

- Load Delay occurs only if necessary (dependent instructions).
- Branch Delay always happens (part of the ISA).
- Why not have Branch Delay interlocked?
  - Answer: Interlocks only work if you can detect hazard ahead of time. By the time we detect a branch, we already need its value … hence no interlock is possible!

Data Stationary Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Review: Forwarding

Fix by Forwarding result as soon as we have it to where we need it:

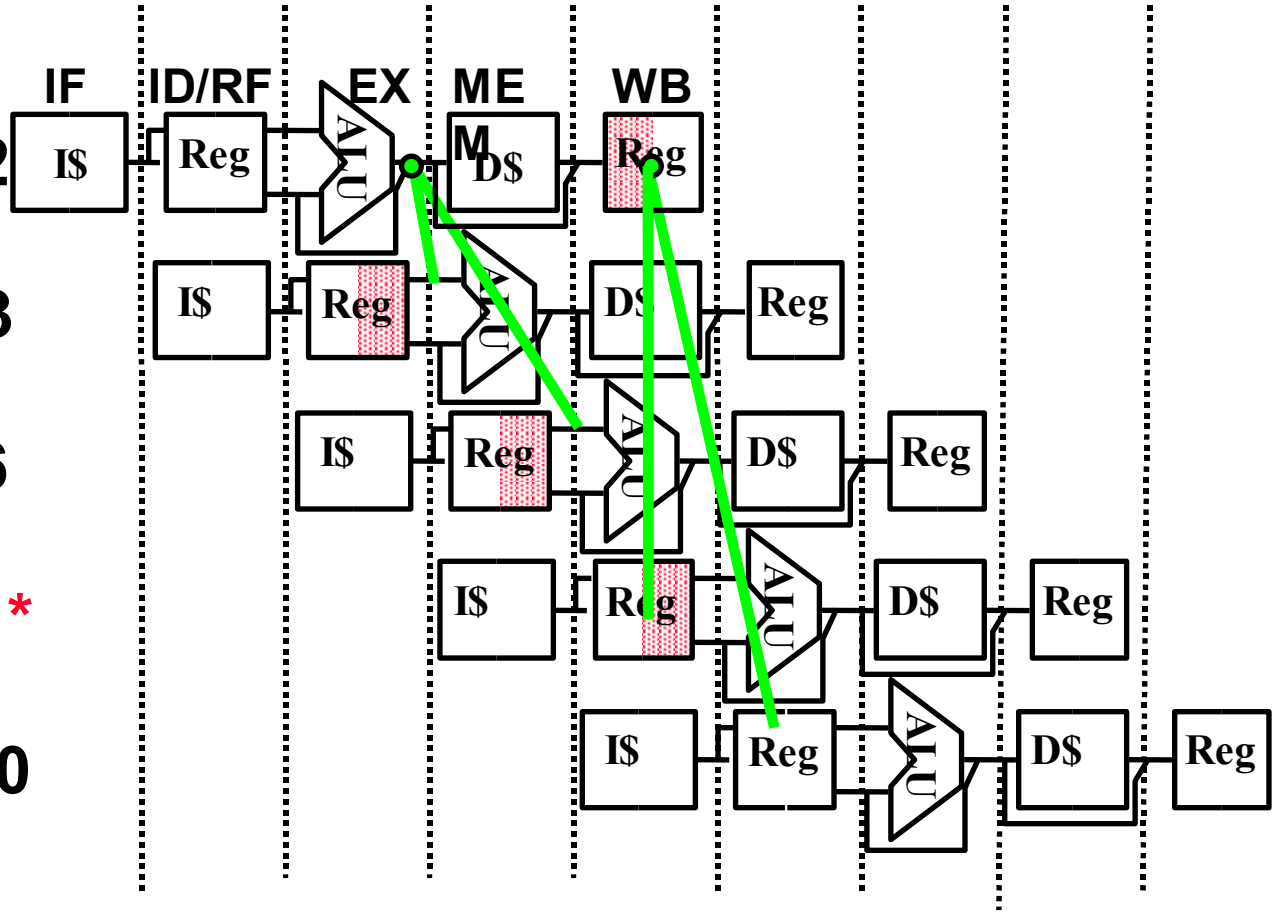
add $t0,$t1,$t2

sub $t4,$t0,$t3

and $t5,$t0,$t6

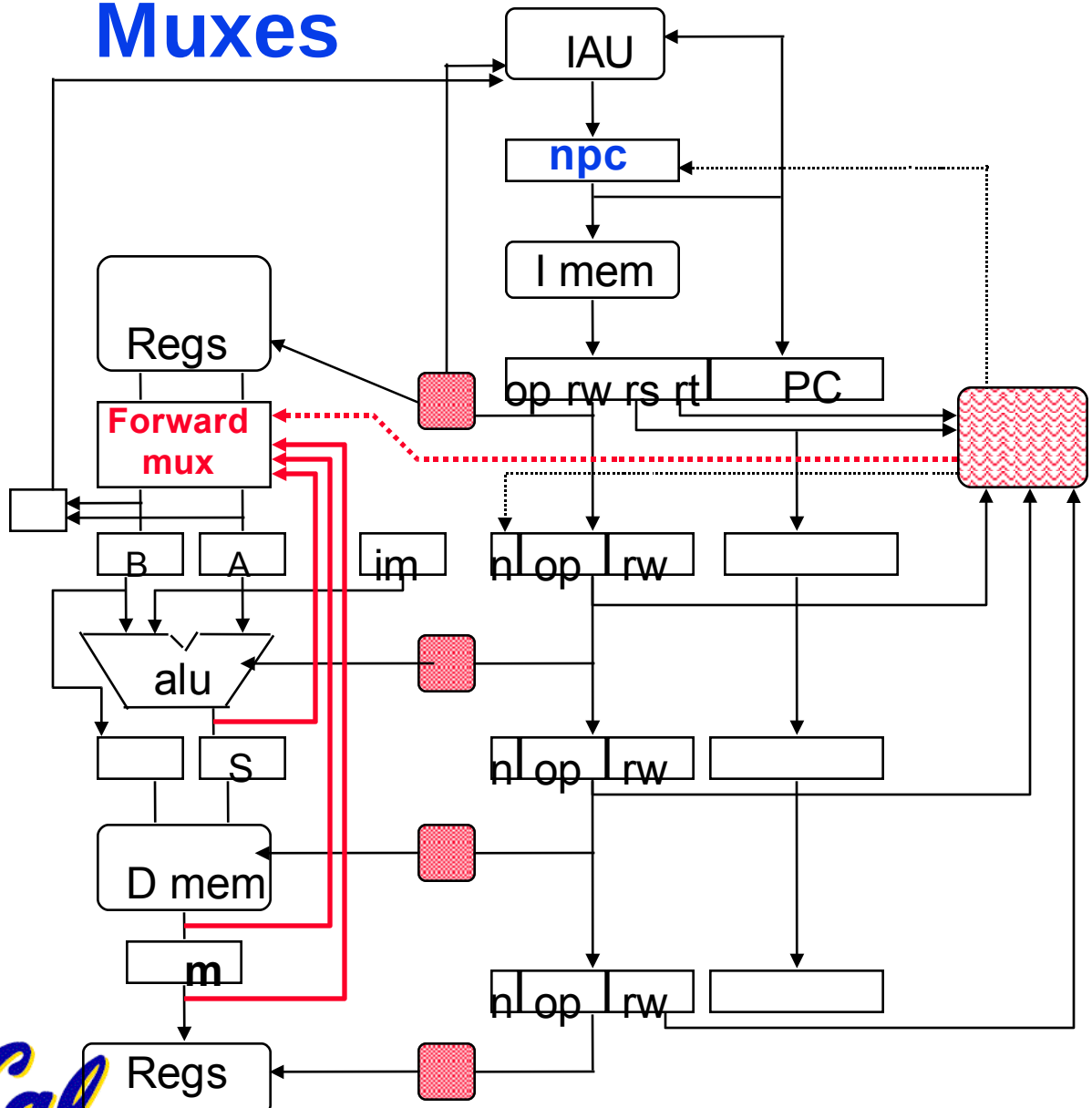
or $t7,$t0,$t8 *

xor $t9,$t0,$t10



* "or" hazard solved by register hardware

Forwarding Muxes



- Detect *nearest* valid write op operand register and forward into op latches, bypassing remainder of the pipe
- Increase muxes to add paths from pipeline registers
- Data Forwarding = Data Bypassing