pollev.com/kakiryan

COMP311: *COMPUTER ORGANIZATION!*

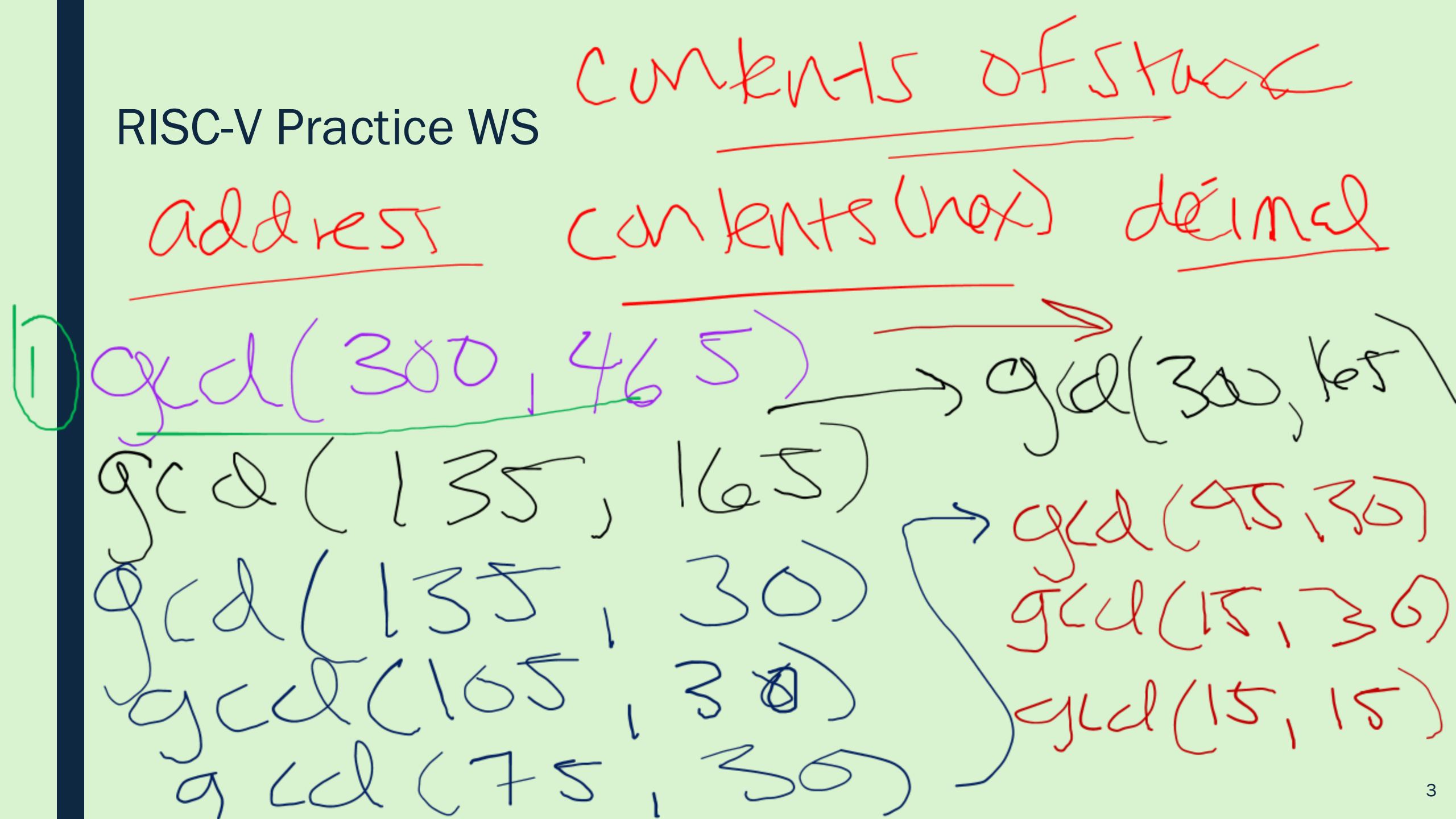
Lecture 24: Pipelining the RISC-V CPU, Pipeline Hazards

tinyurl.com/comp311-fa25

Quiz Topics

- RISC-V Assembly!
  - *How to read + write basic programs in assembly*
  - *How to translate from C/C pseudocode to assembly, and back*
  - *Understanding the RISC-V memory layout, the stack, registers & how they all work together*
- The full RISC-V Datapath
  - *Conceptually, how do all the different instruction types work*
- The best resource to study will be the in-class problems we have done together. The WS problems are harder than what will be on the quiz!

RISC-V Practice WS



```
reset:
0:   lui    sp,0xc0000
4:   addi   sp,sp,0xff0
8:   jal    ra,main        # RA = 12
12:  j      halt

gcd:
16:  addi   sp,sp,-4
20:  sw     ra,(sp)
24:  beq    a0,a1,return
28:  blt    a0,a1,else
32:  sub    a0,a0,a1
36:  jal    ra,gcd         # RA = 40
40:  beq    x0,x0,return

else:
44:  sub    a1,a1,a0
48:  jal    ra,gcd         # RA = 52

return:
52:  lw     ra,(sp)
56:  addi   sp,sp,4
60:  ret

y:
64:  .word  0

main:
68:  addi   sp,sp,-4
72:  sw     ra,(sp)
76:  li     a0,300
80:  li     a1,465
84:  jal    x1,gcd         # RA = 88
88:  sw     a0,y
92:  lw     ra,(sp)
96:  addi   sp,sp,4
100: ret
```

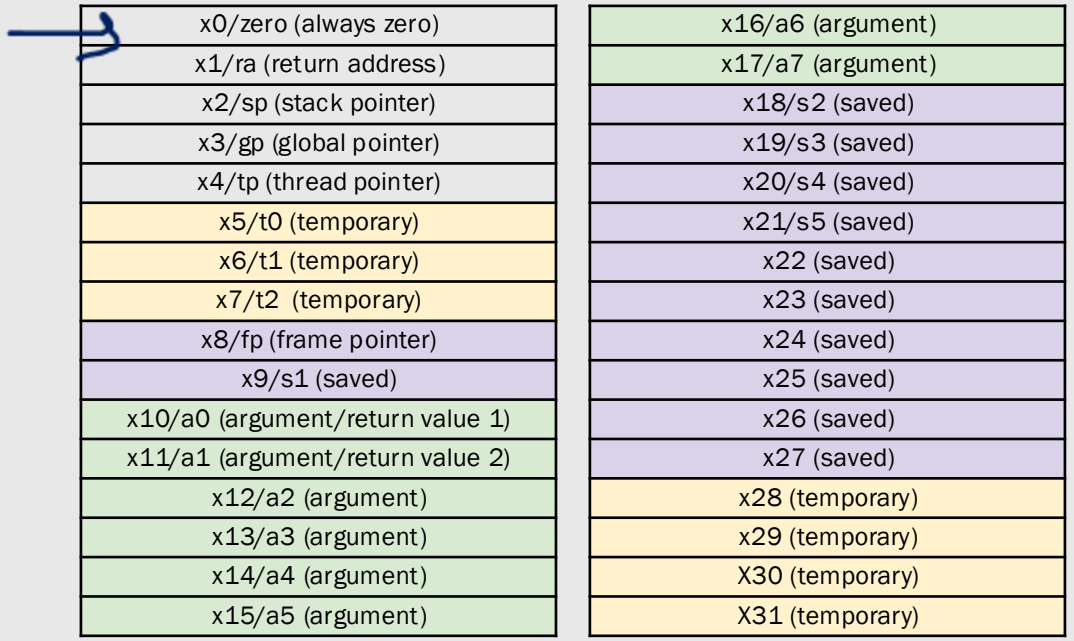
THE APPLICATION BINARY INTERFACE (ABI) AND PROCEDURE CALLS!

Register use conventions

By convention, the RISC-V registers are assigned to specific uses and names used in the ABI. These are supported by the assembler, and high-level languages.

We'll use these names increasingly.

Why have such conventions?



x0/zero (always zero)	x16/a6 (argument)
x1/ra (return address)	x17/a7 (argument)
x2/sp (stack pointer)	x18/s2 (saved)
x3/gp (global pointer)	x19/s3 (saved)
x4/tp (thread pointer)	x20/s4 (saved)
x5/t0 (temporary)	x21/s5 (saved)
x6/t1 (temporary)	x22 (saved)
x7/t2 (temporary)	x23 (saved)
x8/fp (frame pointer)	x24 (saved)
x9/s1 (saved)	x25 (saved)
x10/a0 (argument/return value 1)	x26 (saved)
x11/a1 (argument/return value 2)	x27 (saved)
x12/a2 (argument)	x28 (temporary)
x13/a3 (argument)	x29 (temporary)
x14/a4 (argument)	X30 (temporary)
x15/a5 (argument)	X31 (temporary)

Prevents functions from accidentally overwriting each other's values

Allows independently compiled code to co-execute correctly!

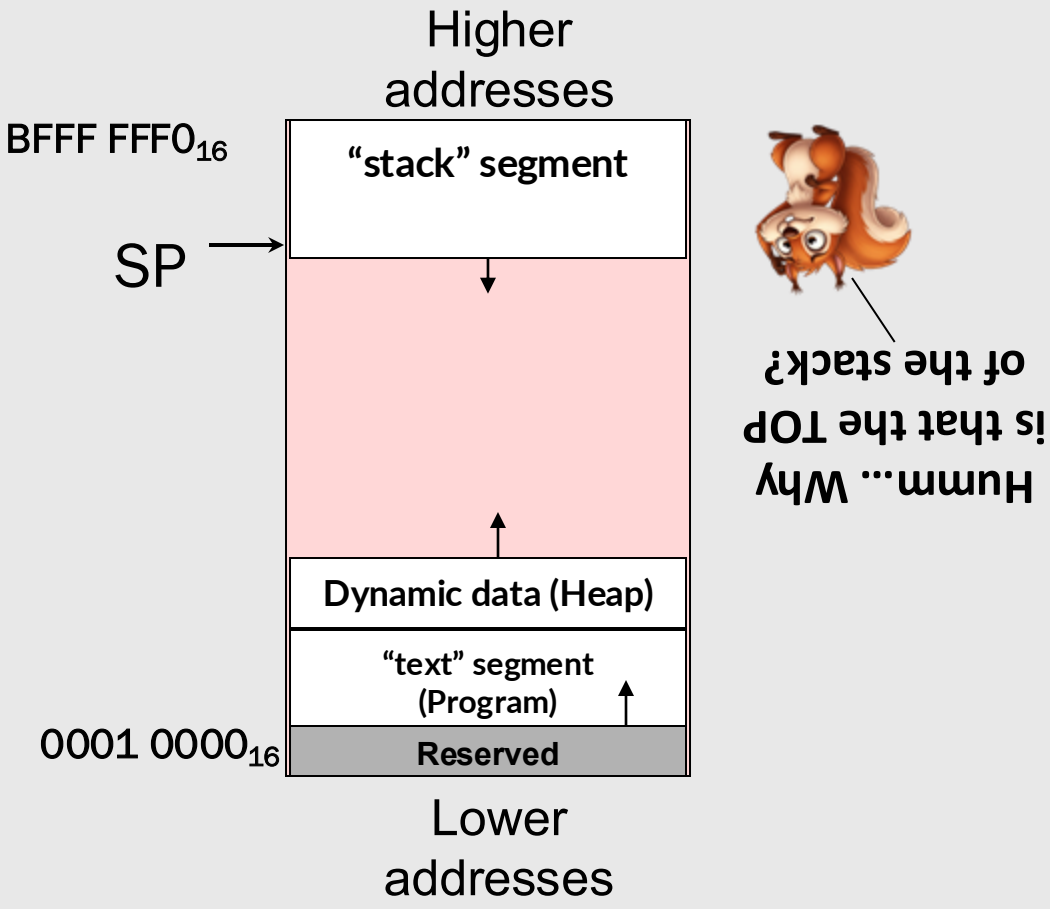
Separately written fns can safely share CPU without corrupting each other's data

RISC-V Stack Convention

These conventions for allocating variables when calling fns → part of the ABI

CONVENTIONS:

- **Assign a register as the Stack Pointer (sp = x2).**
- **Stack grows DOWN (towards lower addresses) on *pushes* and *allocates***
- ***sp* points to the last or TOP *used* location.**
- **Stack is placed far away from the program and its data.**



Stack Management Policies

ALLOCATE k: reserve k WORDS of stack

SP = SP - 4*k

```
addi    sp,sp,-4*k
```

DEALLOCATE k: release k WORDS of stack

SP = SP + 4*k

```
addi    sp,sp,4*k
```

PUSH x: push Reg[x] onto stack

Mem[SP - 4] = Rx

SP = SP - 4

```
addi    sp,sp,-4
sw      rx,(sp)
```

POP x: pop the top of the stack into Reg[x]

Rx = Mem[SP]

SP = SP + 4

```
lw      rx,(sp)
addi    sp,sp,4
```

Incorporating A Stack

```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}


main()
{
  sqr(10);
}
```

```
sqr:    addi    sp,sp,-8
        sw      ra,4(sp)
        sw      s0,0(sp)
        slti    t0,a0,2
        bne     t0,x0,return
        add     s0,x0,a0
        addi    a0,a0,-1
        jal     ra,sqr
        add     a0,a0,s0
        add     a0,a0,s0
        addi    a0,a0,-1
return: lw      s0,0(sp)
        lw      ra,4(sp)
        addi    sp,sp,8
        jalr    x0,ra

main:   addi    sp,sp,-4
        sw      ra,(sp)
        addi    a0,x0,10
        jal     ra,sqr
        lw      ra,(sp)
        addi    sp,sp,4
        jalr    x0,ra
```

function prologue

function epilogue

Every call pushes a frame (prologue) and pops it (epilogue), so nested/recursive calls don't overwrite each other's saved ra or saved registers

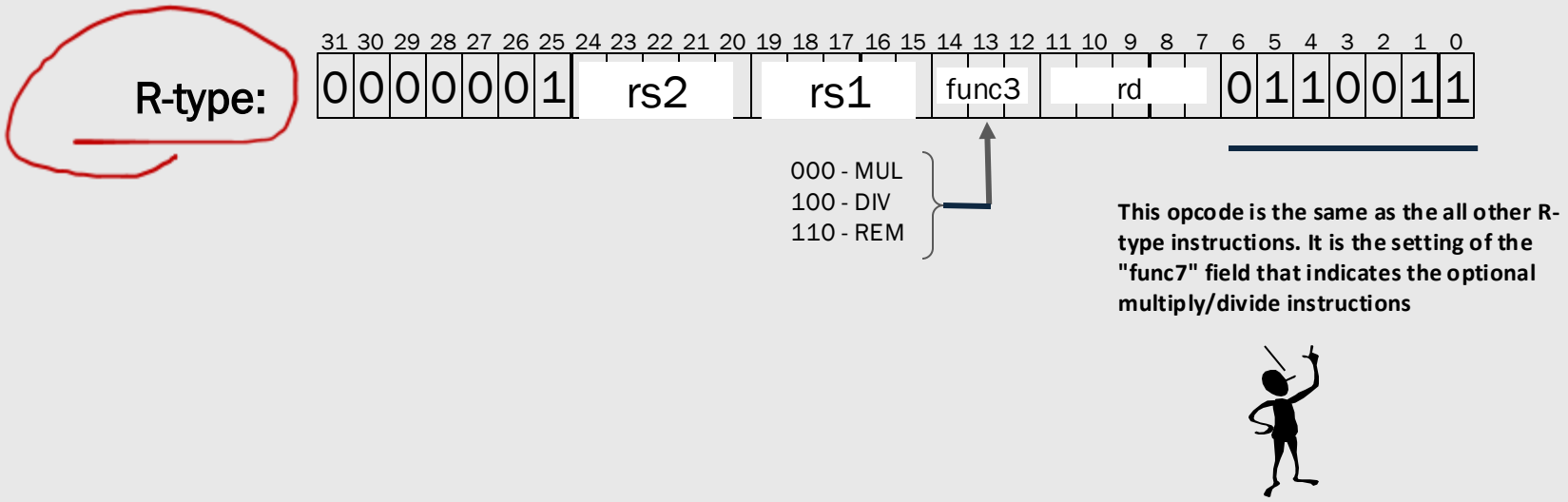
The RISC-V was designed so every instruction keeps the same few fields (opcode, rs1, rs2, rd) in the same positions, and the immediates have to bend around that fixed layout.

MISC. INSTRUCTIONS & FUN IMMEDIATE ENCODING DETAILS ☺

The immediates are complicated/confusing in some cases b/c RISC-V prioritizes a consistent instruction layout over simple immediate encoding! (design tradeoff…)

Missing Math instructions

The RISC-V ISA includes integer multiplication and division as an extension to the RV32I minimal ISA called RV32M. This is because multiply and divide require significant additional H/W. These instructions can always be emulated, and cost is a consideration for embedded systems. Our miniRISCV simulator includes a subset of RV32M, the subset necessary to implement C.



Multiply

MUL: multiply registers

Syntax: **mul *rd*,*rs*,*rt***

Encoding: **0000 001*t tttt ssss s*000 *dddd d*011 0011**

Description:

Reg[d] ← Reg[s] * Reg[t]

Multiply the contents of *Reg[s]* and *Reg[t]*, and place the lower 32-bits of the product in *Reg[d]*. ***Overflows are ignored.*** MUL is binary and semantically compliant with the RV32M mul instruction.

Example: `mul x5,x6,x7` `# Encoded as: 0x027302B3`

Divide

DIV: divide registers

Syntax: **div *rd,rs,rt***

Encoding: **0000 001*t tttt ssss s*100 *dddd d*011 0011**

Description:

Reg[d] ← Reg[s] / Reg[t]

Divide the contents of *Reg[s]* by *Reg[t]*, and place the quotient in *Reg[d]*. ***A divisor of 0 does not generate an overflow***, and the destination register *rd* is set to all ones. DIV is binary and semantically compliant with the RV32M div instruction.

Example: **`div x7,x5,x6 # Encoded as: 0x0262C3B3`**

Remainder

REM: remainder of a register quotient

Syntax: **rem *rd,rs,rt***

Encoding: **0000 001*t tttt ssss s*110 *dddd d*011 0011**

Description:

Reg[d] ← Reg[s] % Reg[t]

Divide the contents of *Reg[s]* by *Reg[t]*, and place the remainder in *Reg[d]*. A divisor of 0 does not generate an overflow and the contents of the destination register *rd* is set to the dividend, *Reg[s]*. REM is binary and semantically compliant with the RV32M rem instruction.

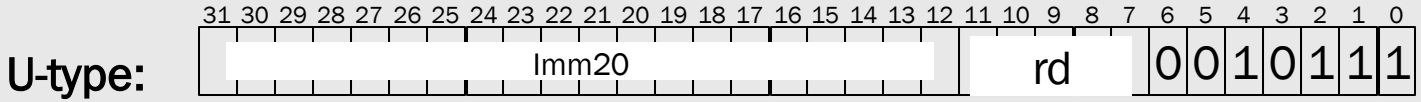
Example: **`rem x7,x5,x6  # Encoded as: 0x0262E3B3`**

Another instruction: AUIPC

PC-relative branch + jump instructions allow for relocatable code. However, what about relocatable “data”?

Ex: you might want to place a data structure in a code section and be able to relocate it without the need for keeping track of the absolute addresses.

RISCV fixes this problem with a PC-relative **lui-like** instruction called **auipc**




```
auipc x10,next  # encoded as 0x00000517
```

AUIPC details

This instruction gives you a PC-relative base address. Once you have it in a register (rd), you can use a load instruction with it to read memory

AUIPC: Add Upper Immediate to PC

Syntax: **auipc *rd,imm20***
Encoding: ***iiii iiii iiii iiii iiii dddd d*011 0011**
Description:

Reg[d] ← PC + sign_extend(imm20 << 12)
Adds the sign-extended 20-bit immediate value, left-shifted by 12 bits, to the program counter, and writes the result to Reg[d].

Example: **`auipc x31,1024`** **`# Encoded as: 0x00400F97`**

Also, the combination of an **auipc** instruction and a **jalr** can transfer control (jump) to any memory location. Both branch and jump instructions have limited ranges.

Load/Store offsets are only 12 bits, we can't encode large addresses directly. AUIC shifts a 20 bit immediate by 12 to get the upper 20 bits of an address near our target.

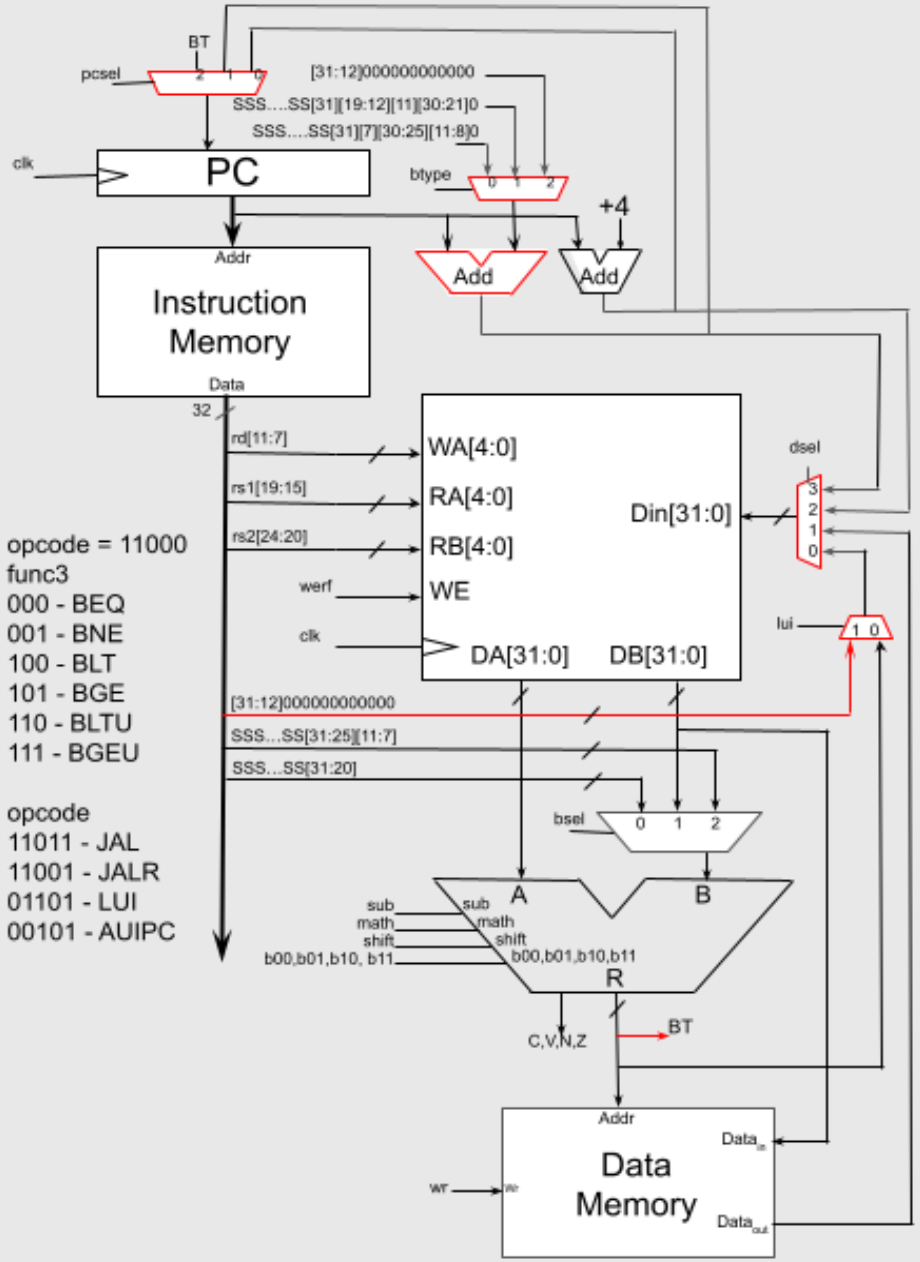
AUIPC + load offset = full 32 bit address computation

Branch, Jump, lui and auipc

Branches add a 12-bit “sign-extended” immediate value (multiplied by 2) to the current PC if taken.

Jumps always add a 20-bit sign-extended immediate value and support saving PC+4.

We also need to be able to compute the PC-relative offsets used by the AUIPC instruction, and large immediates of LUI and store them into the register file.

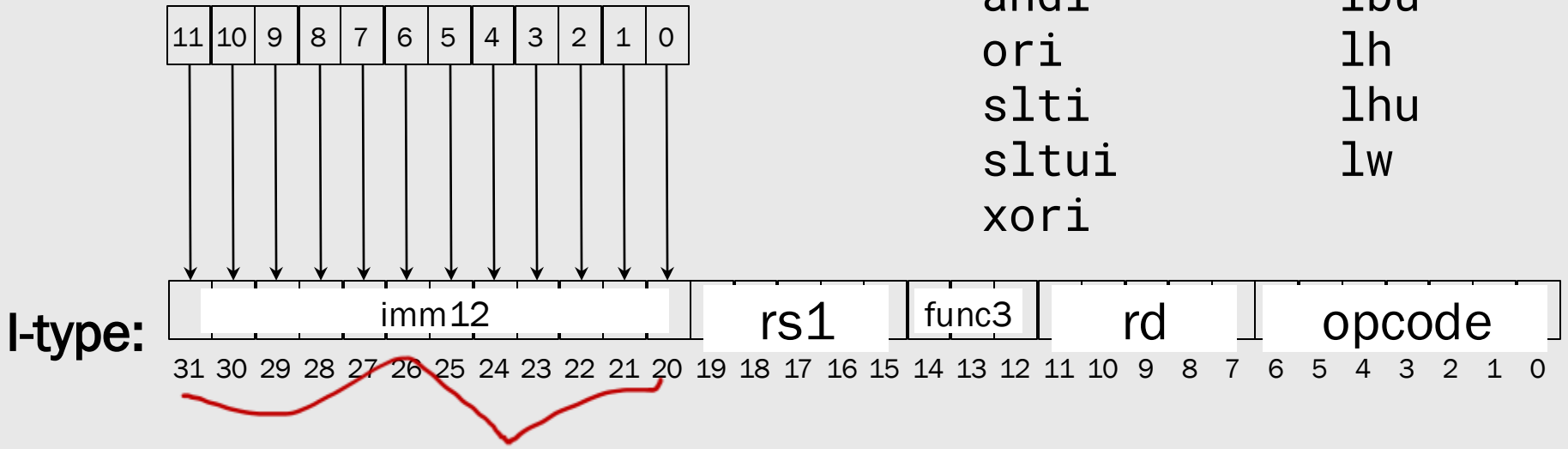


Immediate Encodings

Many RISC-V instructions encode a constant value as part of the instruction. Unlike other ISAs, the encoding of constants is not uniform. The I-type instructions encode their immediate operand as follows:

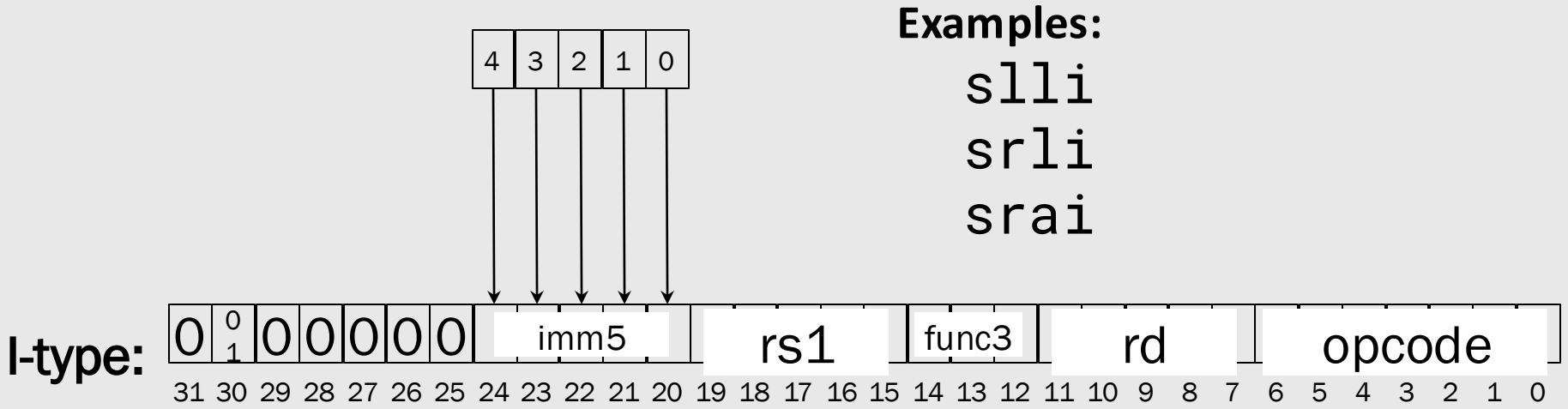
Examples:

```
addi        lb
andi        lbu
ori         lh
slti        lhu
sltui       lw
xori
```



Immediate Encodings

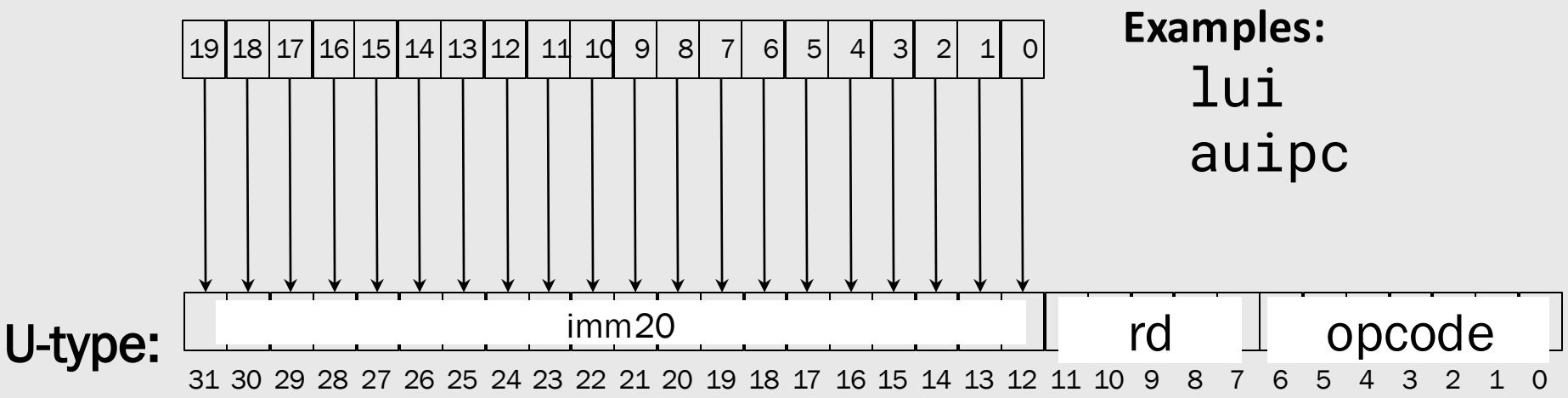
Many RISC-V instructions encode a constant value as part of the instruction. Unlike other ISAs, the encoding of constants is not uniform. The I-type instructions encode their immediate operand as follows:



But there are exceptions!

LUI's Immediate Value

The immediate-field encoding of `lui` and `auipc` also seems straightforward, but, the immediate values might not be what you expect due to the sign-extension of its companion addi instruction.



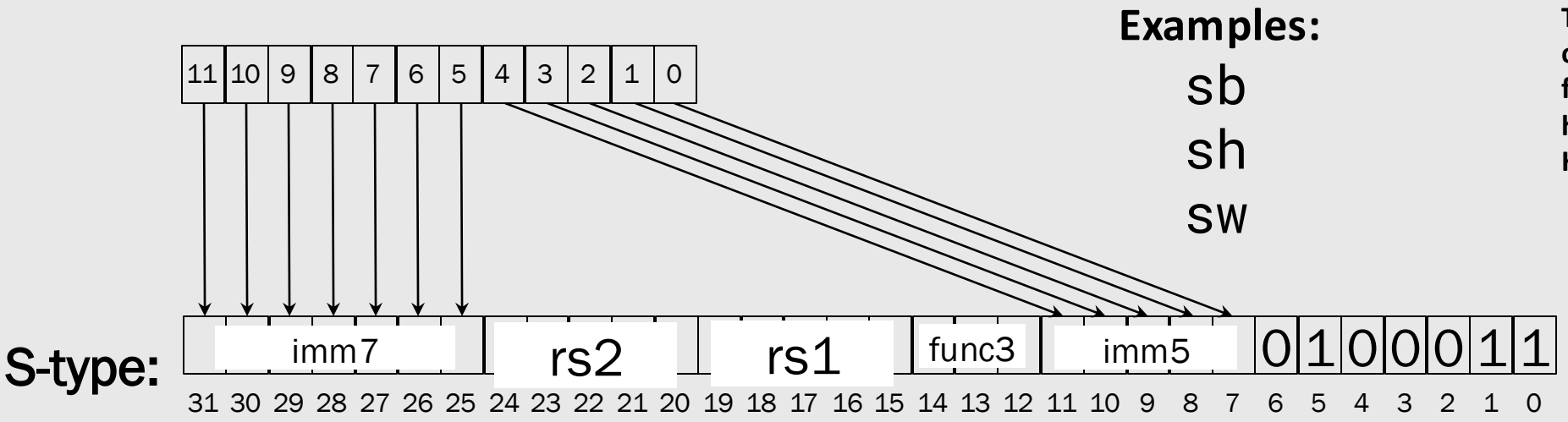
Examples:

```
lui
auipc
```

if bit 11 of the large constant is set, then you add 1 to the LUI/AUIPC immediate value.

Immediate Offsets for Stores

RISC-V store instructions include an offset, but it is not encoded in an obvious way. Note that there is no need for a rd register in a store, but both rs1 and rs2 are needed if the encodings are used consistently.



Examples:

sb

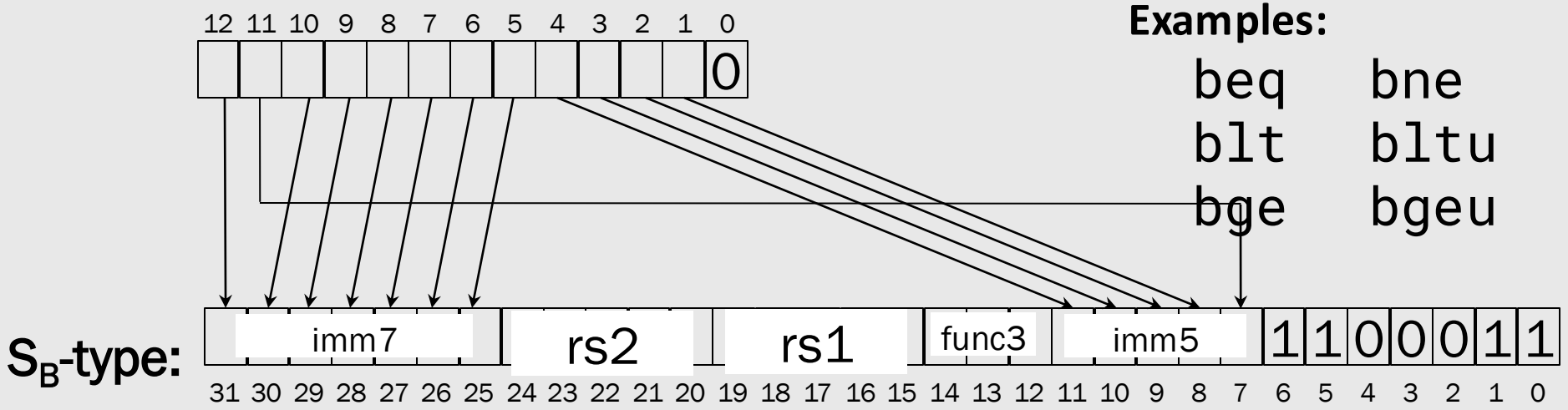
sh

sw

The assembler will take care of filling in the immediate fields, of the instruction. It has some impact on the H/W design

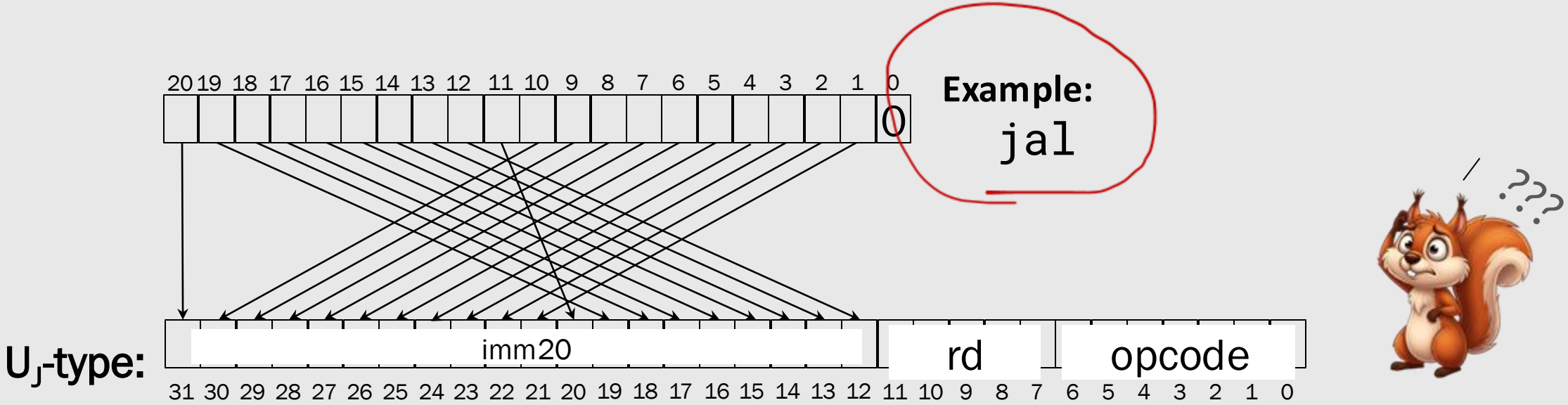
Immediate Offsets for Branches

The offsets of RISC-V branch instructions use the same format as a stores, but the ***offset encoding*** is non-obvious. Again, there is no need for a rd register; only rs1 and rs2 are used. Branch offsets can only be even. In fact, all instructions must be aligned to half-word boundaries.



Crazy Jump offsets

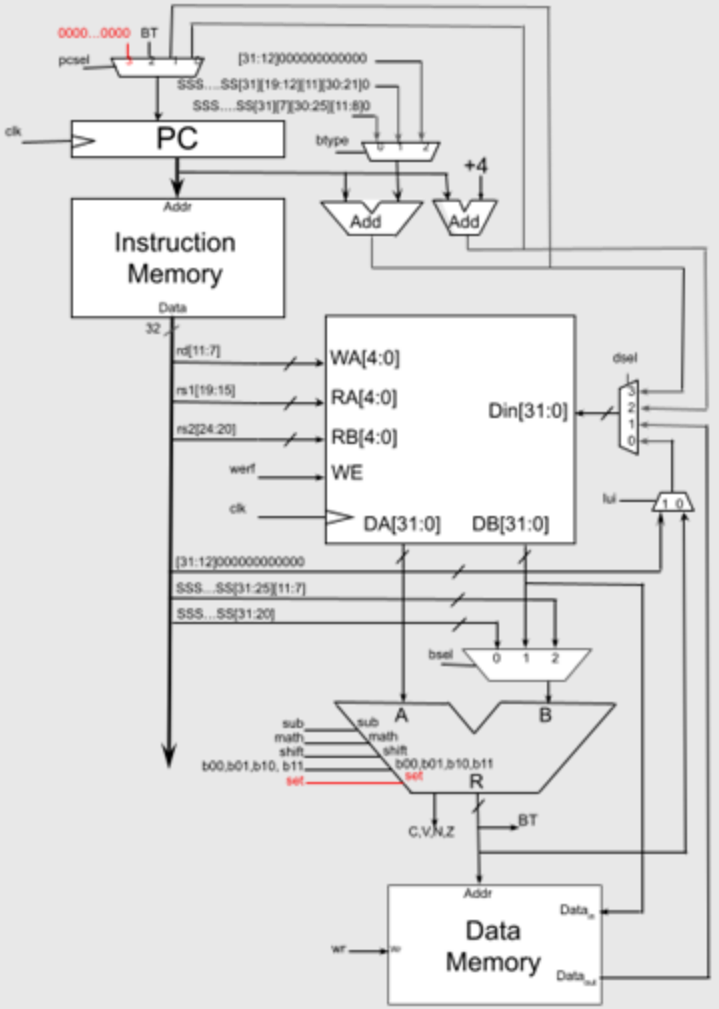
The offsets of RISC-V jump instructions use the same U-format as `lui/auipc`, but the ***offset encoding*** is even stranger. Jump offsets, like branch offsets can only be even.



A Few Loose Ends

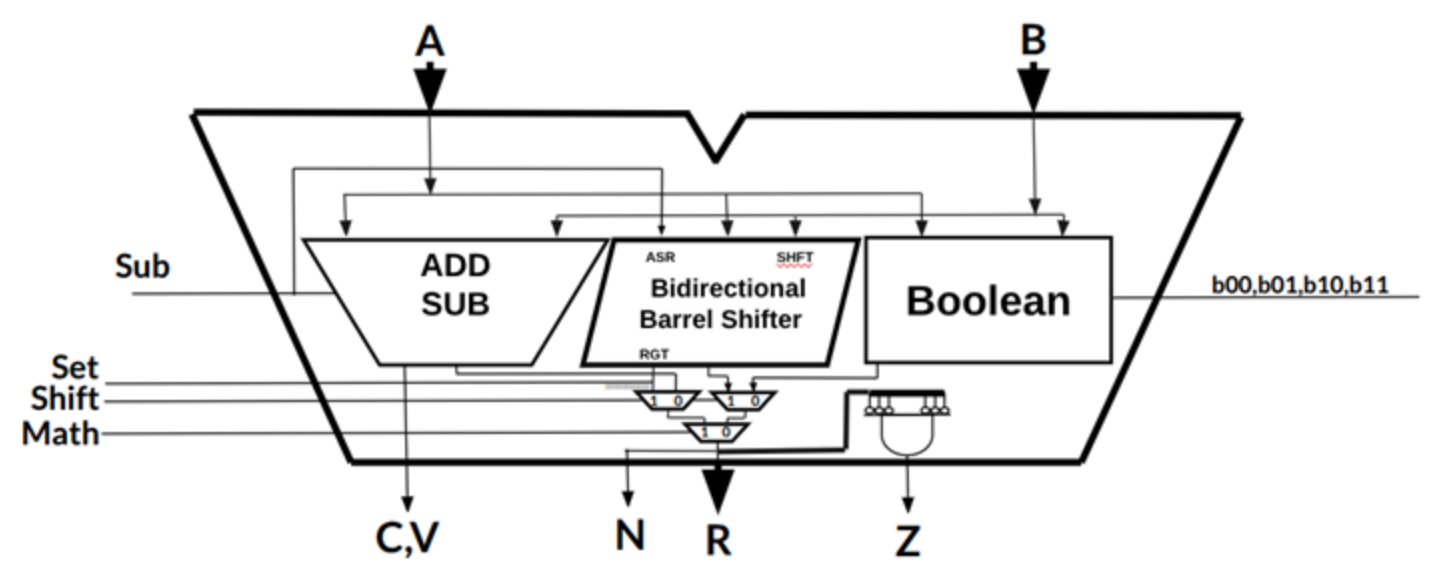
Next we will add a "reset" signal that starts execution at a known address by forcing a value into the PC. Like miniRISCV our cpu starts execution at location 0.

We also need to make a change to the ALU design discussed in Lecture 14



A Modified ALU

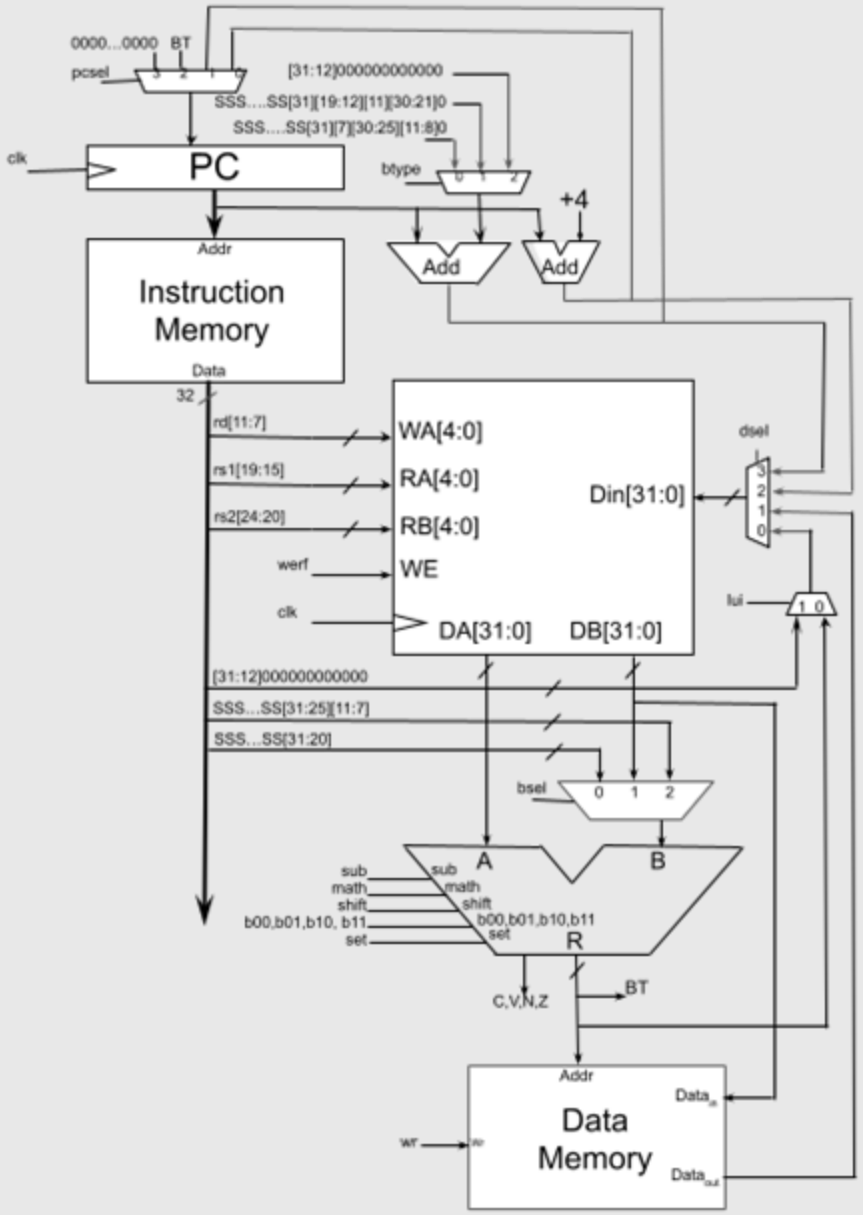
Special support to generate the constants "0" and "1" needed by the SLT, SLTU, SLTI, SLTUI, etc instructions



Math	Shift	Sub	Set	R
0	0	X	X	f(A,B)
0	1	0	0	A << B
0	1	0	1	A >> B
0	1	1	1	A >>> B
1	0	0	X	A+B
1	0	1	X	A-B
1	1	X	0	0x00000000
1	1	X	1	0x00000001

Our full RISC-V Datapath

Signal	Meaning
pcsel	Selects PC+4 vs branch/jump target
BT	Branch taken flag
btype	Chooses B-type immediate formatting
SSS...SS	Immediate extractors
WA	Write register (rd)
RA	Read register A (rs1)
RB	Read register B (rs2)
Din	Data written to register file
DA	Output A from register file
DB	Output B from register file
WE	Register file write enable
dsel	Chooses write-back value (ALU/mem/PC+4/LUI)
bsel	ALU operand B source select
A	ALU operand 1
B	ALU operand 2
ALU control lines	Determine ADD/SUB/AND/OR/shift/SLT/etc.
R	ALU result
C V N Z	ALU flags
wr	Memory write enable
Data_in	Data to memory
Data_out	Data from memory

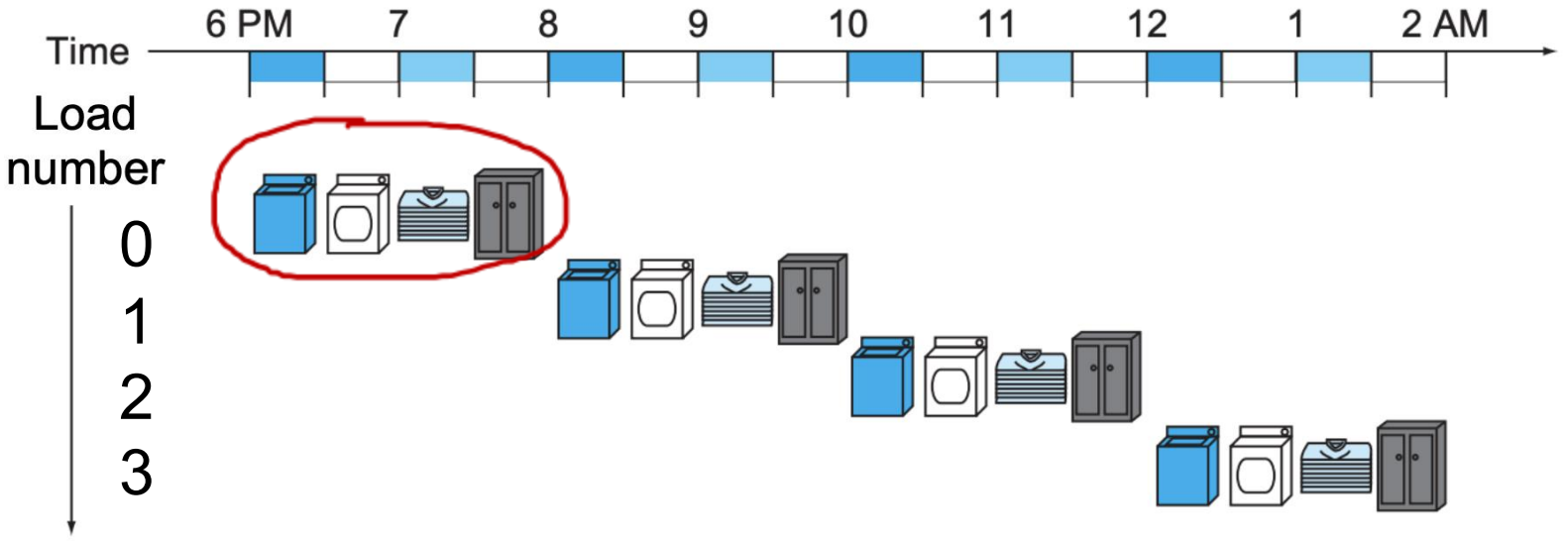


PIPELINING REVIEW!

Single-Cycle Laundry Analogy

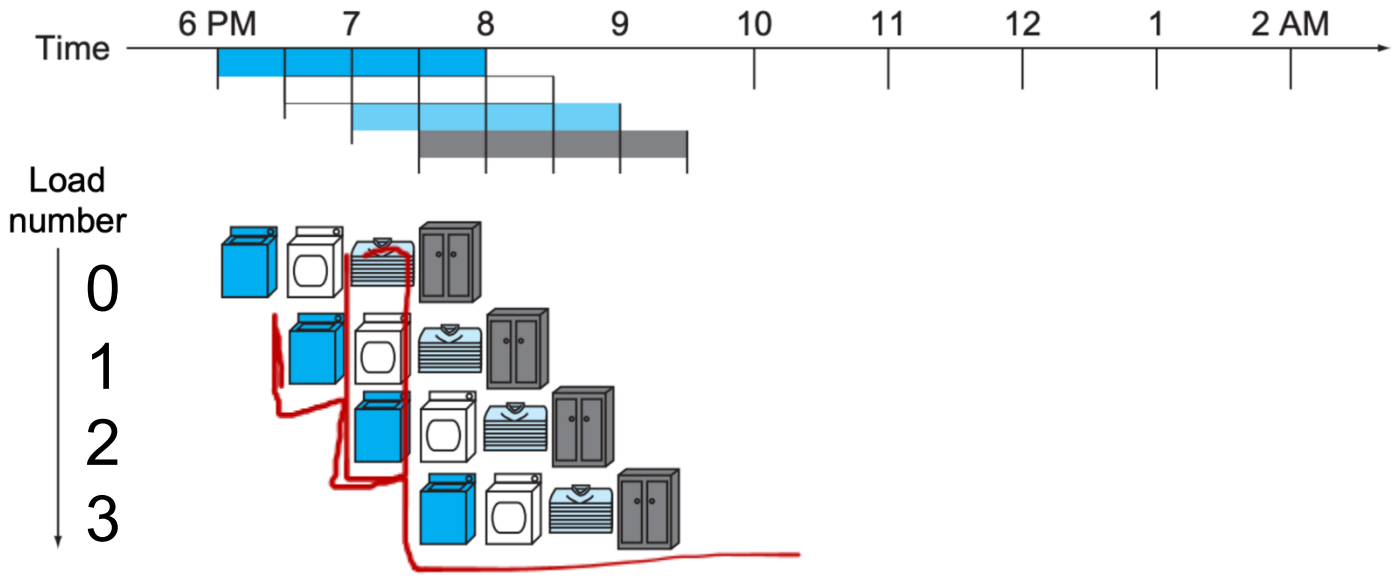
1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on the table and fold.
4. When the folding is finished, ask your roommate to put the clothes away.

When your roommate is done putting the clothes away, start the next load



Pipelined Laundry Analogy

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on the table and fold.
4. When the folding is finished, ask your roommate to put the clothes away.



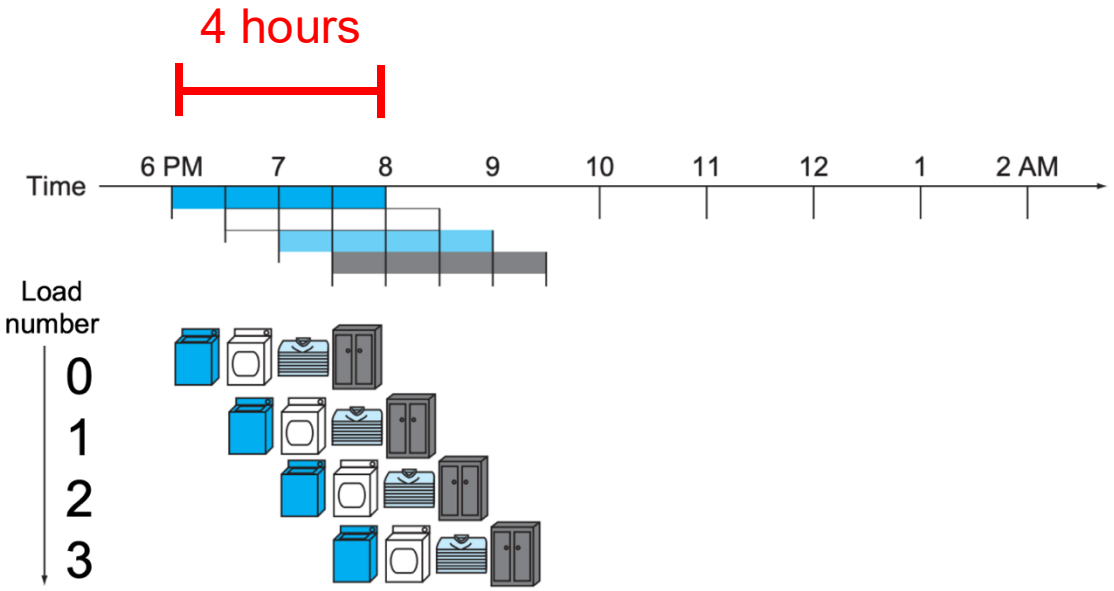
As long as we have separate resources for each stage, we can pipeline the tasks.

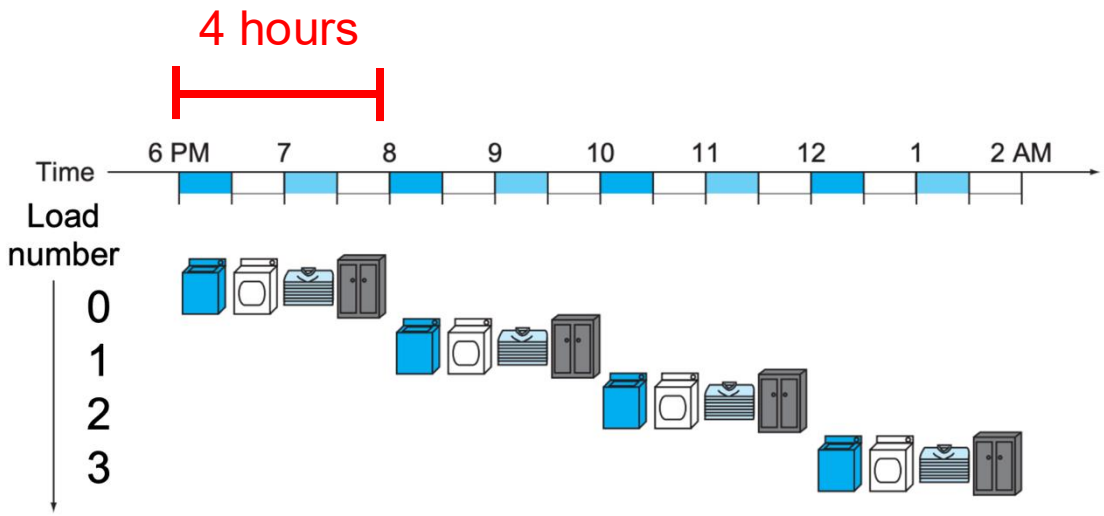
Performance Metrics

- Latency
  - The time it takes to complete an individual load of laundry
- Throughput
  - The number of loads that can be completed per unit time
- Speedup
  - performance improvement achieved by using a pipelined setup vs a non-pipelined setup

Latency

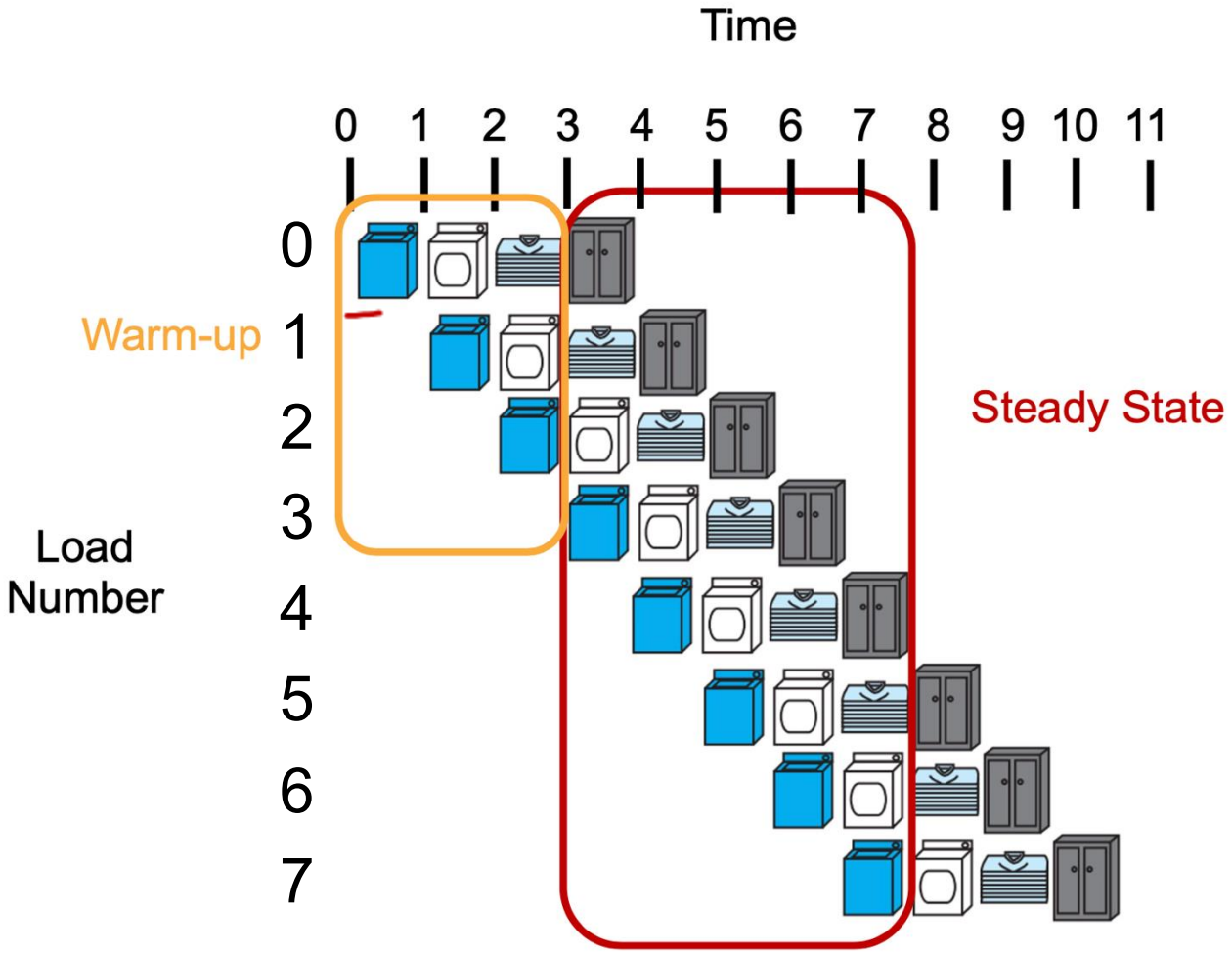
- The latency of a single load of laundry is the same for both the pipelined and non-pipelined approach





Steady-State

- We reach steady state when we are fully utilizing the pipeline



Speedup

- Performance improvement achieved by using a pipelined vs non-pipelined implementation

$$speedup = \frac{execution\ time\ of\ nonpipelined\ implementation}{execution\ time\ of\ pipelined\ implementation}$$

Speedup

- Computing the speedup on 8 loads of laundry

$$speedup = \frac{execution\ time\ of\ nonpipelined\ implementation}{execution\ time\ of\ pipelined\ implementation}$$

$$speedup = \frac{8\ loads\ x\ 4\ hours/load}{11\ hours} = \frac{32}{11} = 2.9$$

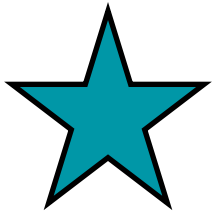
Go two slides back to see where this came from

The pipelined approach is nearly 3x faster than the non-pipelined approach!

Speedup

- As the number of loads increases, what value does the speedup approach?

Speedup

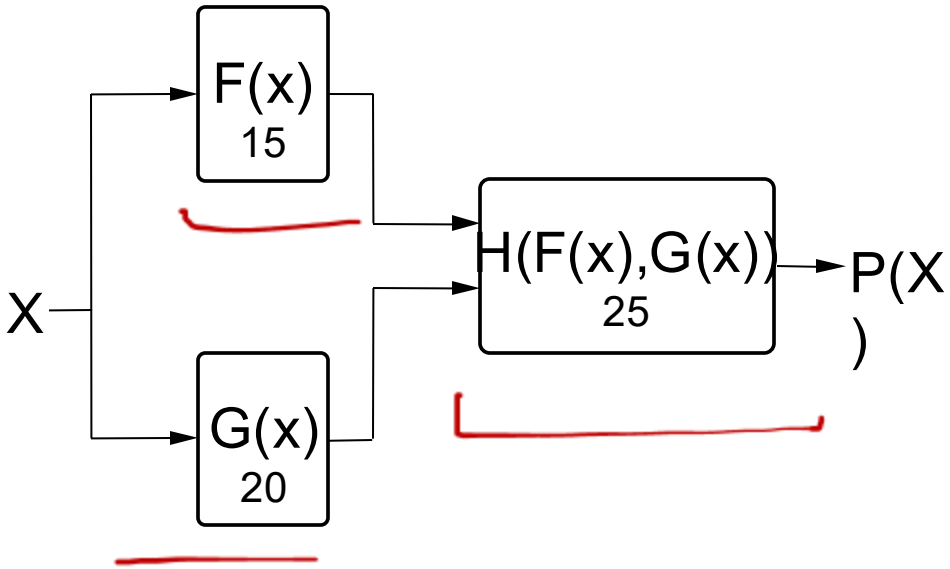


- As the number of loads increases, what value does the speedup approach?

4

With 4 stages, we can be working on at most 4 loads simultaneously

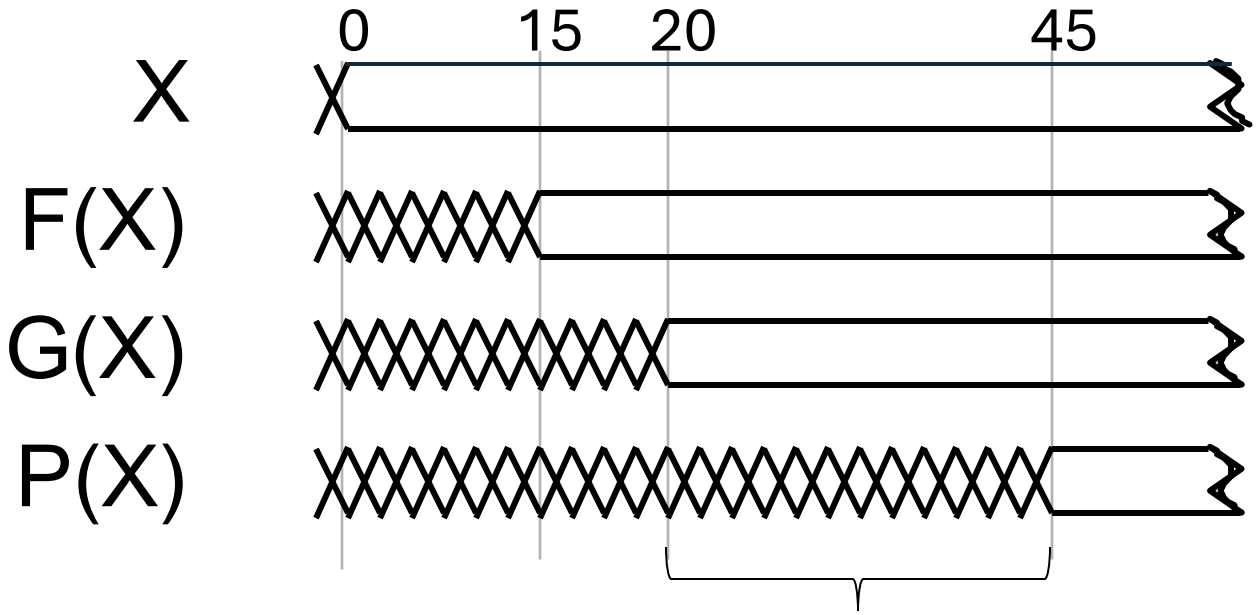
Back to Circuits…



For combinational logic:

- *latency = t_{PD},*
  throughput = $1/t_{PD}$.

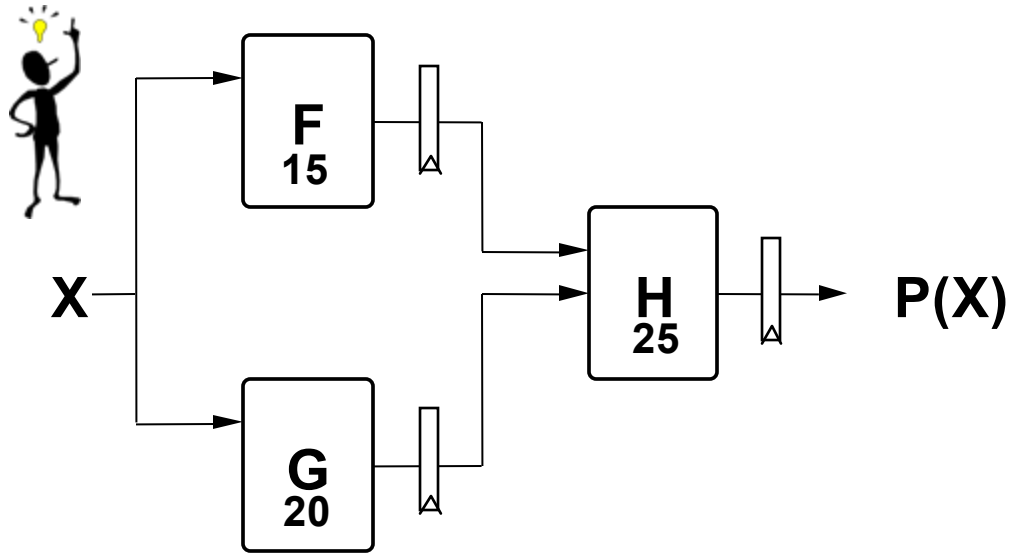
We can't get the answer faster. Are we making effective use of our hardware at all times?



F & G are "idle", just holding their outputs stable while H performs its computation

Pipelined Circuits

Use registers to hold H's inputs stable!



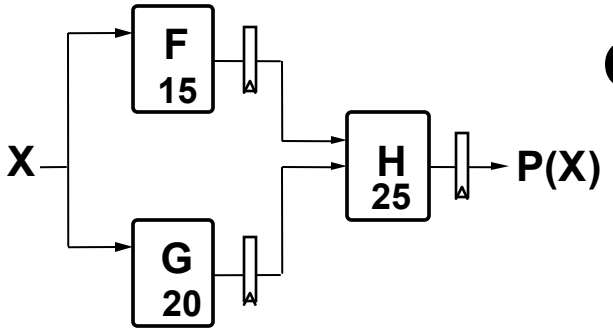
Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ***ideal zero-delay registers*** *(t_{pd} = 0):*

Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage pipeline: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

	latency	*throughput*
unpipelined	**45**	**1/45**
2-stage pipeline	**50**	**1/25**
	worse	***better***

Pipelining uses registers to improve the throughput of combinational circuits

Pipeline Diagrams



Clock cycle

	i	i+1	i+2	i+3	
Input	X_i	X_{i+1}	X_{i+2}	X_{i+3}	…
F Reg		$F(X_i)$	$F(X_{i+1})$	$F(X_{i+2})$	
G Reg		$G(X_i)$	$G(X_{i+1})$	$G(X_{i+2})$	…
H Reg			$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

Pipeline stages

This is an example of parallelism. At any instant we are busy computing 2 results.

A pipeline diagram is just a depiction of what inputs are being processed during a given clock period. The results associated with a particular set of input data move *diagonally* through the diagram, progressing through one pipeline stage on each clock cycle.

Pipeline Conventions

DEFINITION:

A K-Stage Pipeline ("K-pipeline") is an acyclic circuit having exactly K registers on every path from an input to an output.

A COMBINATIONAL CIRCUIT is thus a 0-stage pipeline.

CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its OUTPUTS (as opposed to, alternatively, its inputs).

ALWAYS:

The LATENCY of a K-pipeline is K times the period of the clock common to all registers.

The THROUGHPUT of a K-pipeline is the frequency of the clock.

Pipelining Summary

Advantages:

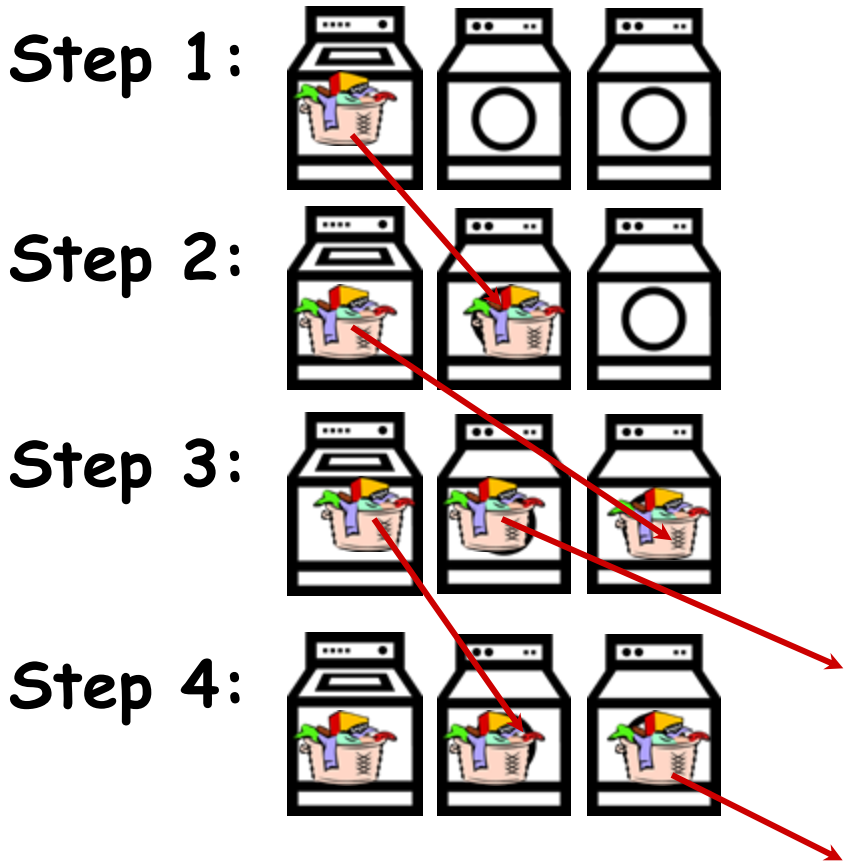
- Higher throughput than combinational system
- Different parts of the logic work on different parts of the problem…

Disadvantages:

- Generally, increases latency
- Only as good as the *weakest* link
  (often called the pipeline's BOTTLENECK)

Is there a way around this "weak link"?

How UNC students REALLY do Laundry?



How to work around a bottleneck.

First, find a place with twice as many dryers as washers.

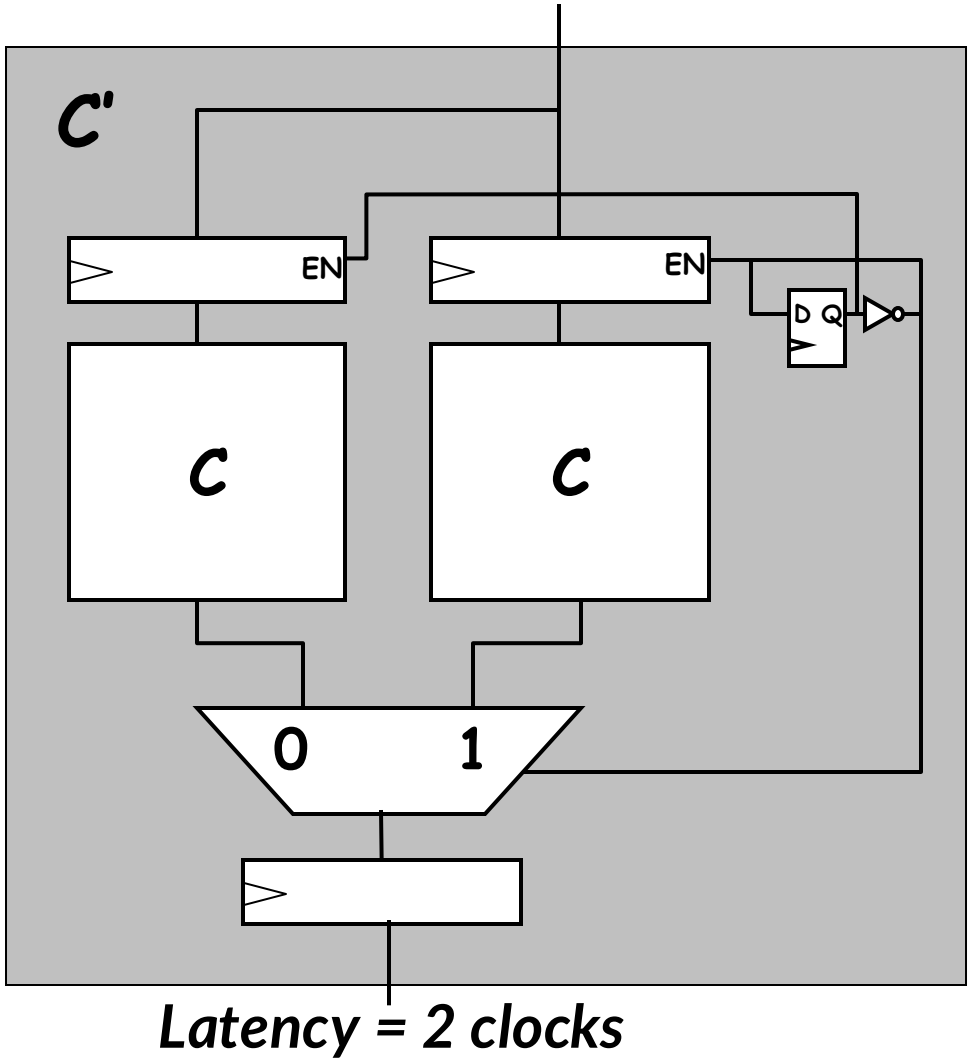
Throughput = $\frac{}{1/30}$ loads/min

Latency = $\frac{}{90}$ mins/load

This Speed-up is called "Interleaving"

One way to overcome a pipeline bottleneck is to **replicate the critical element** as many times as needed and *alternate* inputs between the various copies.

N-way interleaving is equivalent to how many pipeline Stages? __**N**



Latency = 2 clocks

Another Approach… Parallelism

We can combine interleaving and pipelining with parallelism.

Throughput = 2/30 = 1/15 load/min

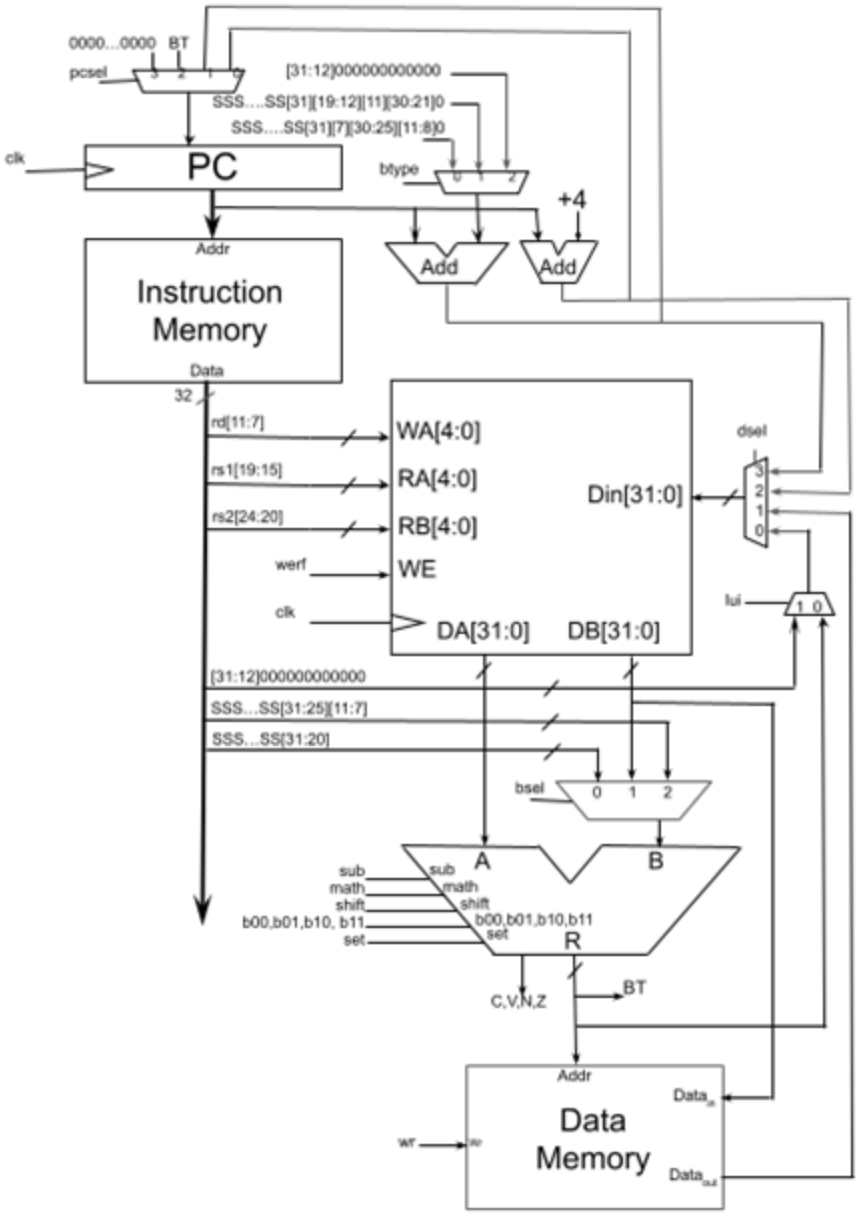
Latency = 90 min

How to Pipeline This?

A CPU is a digital circuit like any other. Thus, we should be able to pipeline it to increase its throughput.

However, there are a few tricky issues.

- It already has registers that get updated on each clock (register file and PC)
- It has feedback, the ALU or Data Memory outputs are routed back to the register file



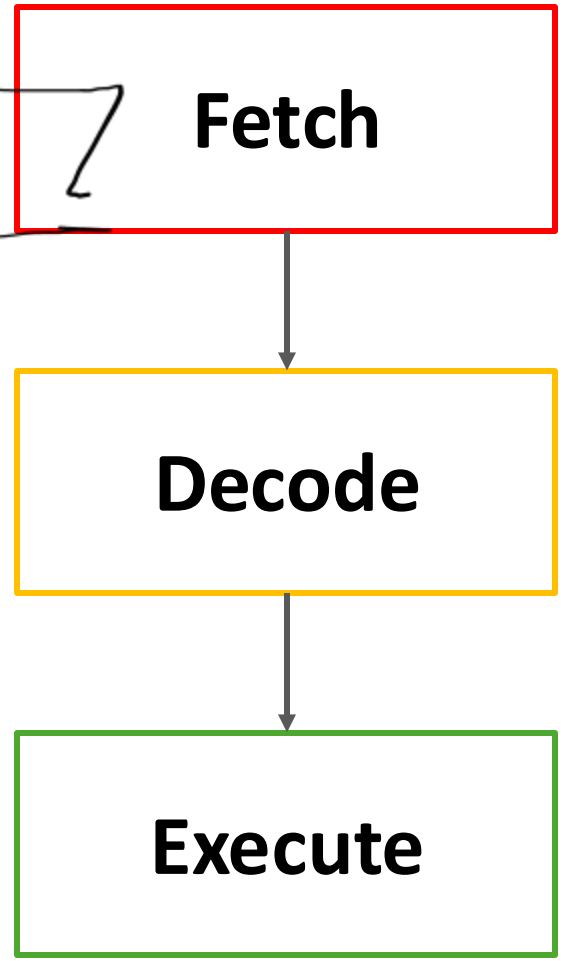
Our Goal

A simple 3-stage ~~pipeline~~:

Fetch:
Instruction memory access

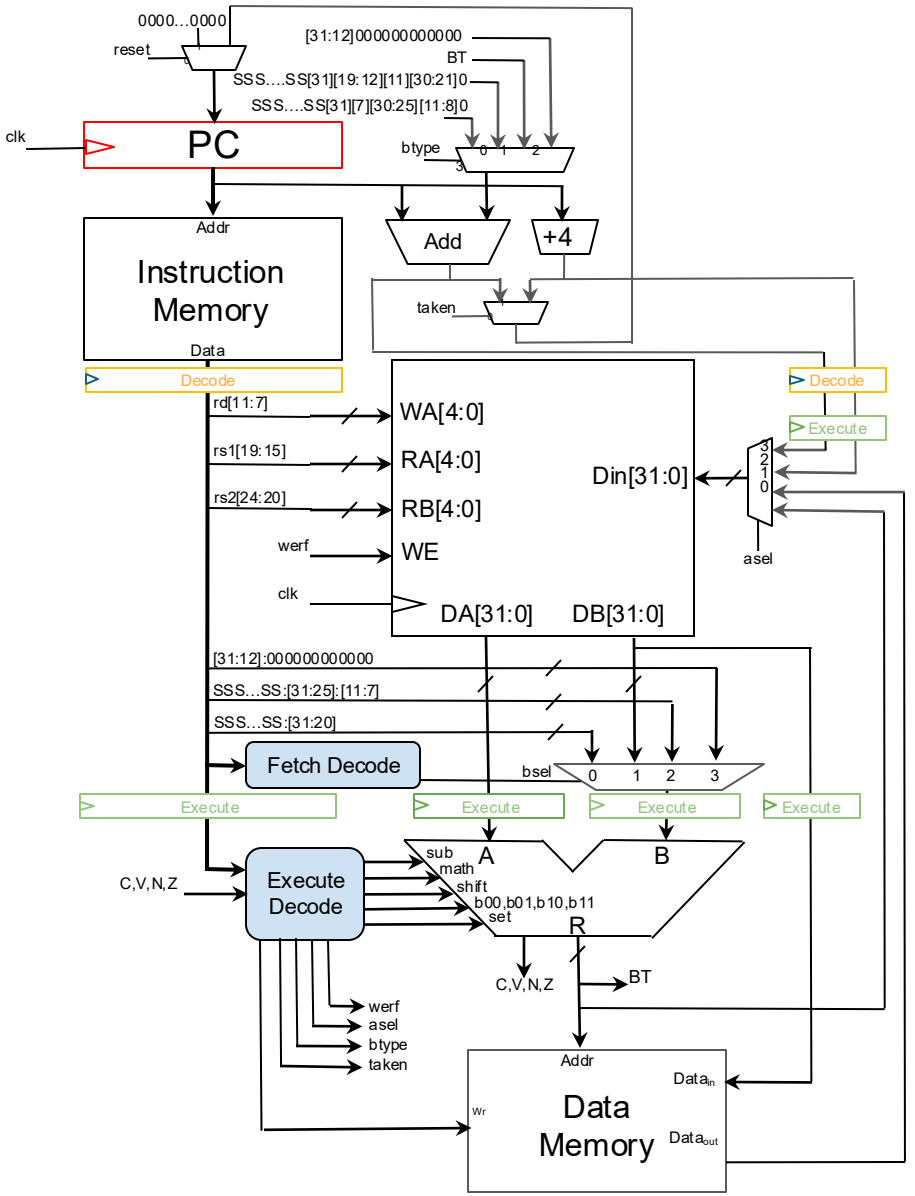
Decode:
Decode instruction
Get source register operands

Execute:
ALU operation
Write-back destination register



Pipelining the Datapath!

- Fetch, Decode, and Execute Stages
- Instructions are decoded in both the Fetch and Decode stages
- Register ports are “Read” in the Decode stage and “Written” at in the end of
  the Execute stage
- PC+4, and PC relative calculations are “delayed”
  for use in later stages

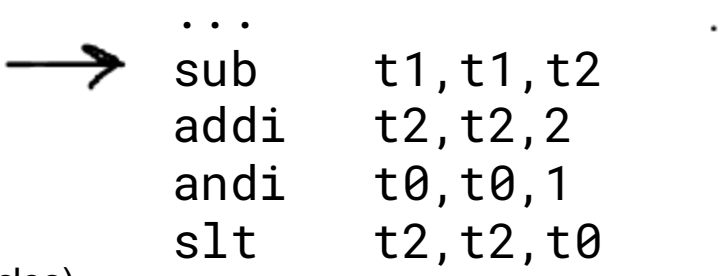


How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.



Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub t1,t1,t2					
Decode						
Execute						

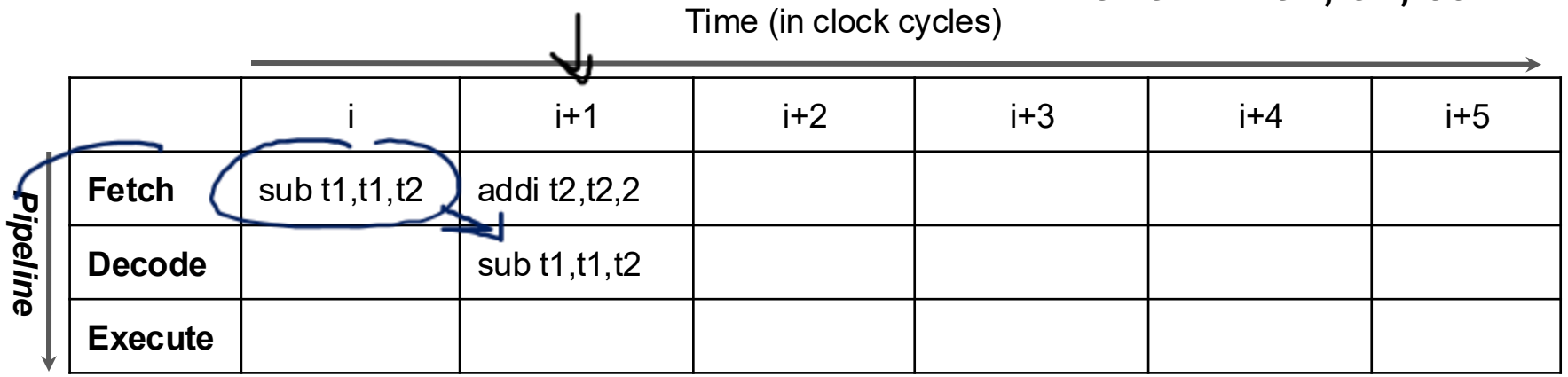
How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub    t1,t1,t2
addi   t2,t2,2
andi   t0,t0,1
slt    t2,t2,t0
```



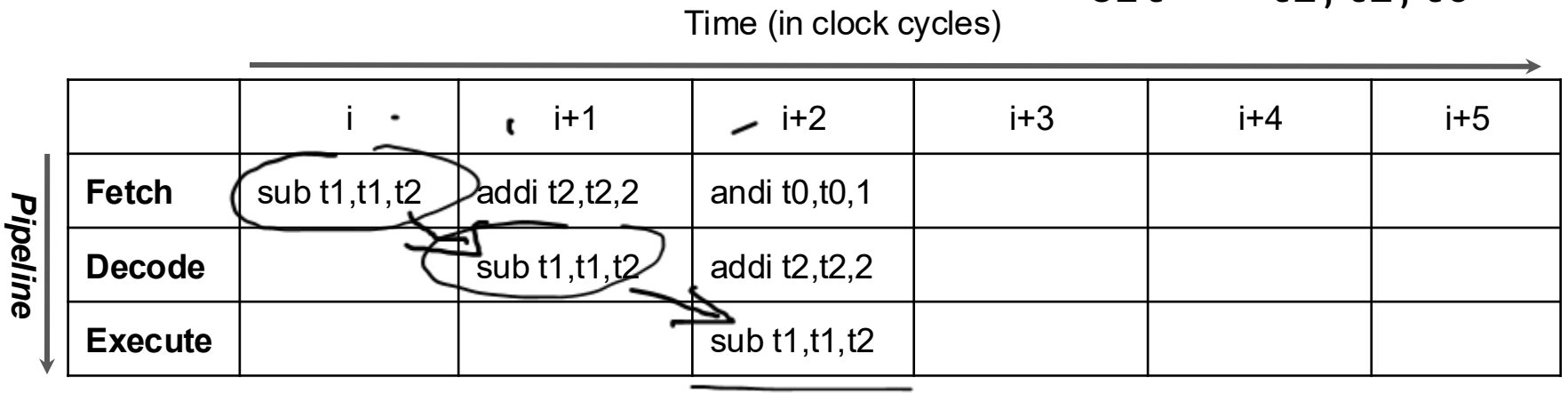
How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub    t1,t1,t2
addi   t2,t2,2
andi   t0,t0,1
slt    t2,t2,t0
```



How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub     t1,t1,t2
addi    t2,t2,2
andi    t0,t0,1
slt     t2,t2,t0
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0		
Decode		sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0	
Execute			sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0

Source operands are fetched in this stage

How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

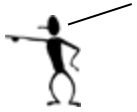
Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub     t1,t1,t2
addi    t2,t2,2
andi    t0,t0,1
slt     t2,t2,t0
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0		
Decode		sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0	
Execute			sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0



Source operands are fetched in this stage

Destination operands are updated in this stage

Simple Instruction flow

Consider the following instruction sequence:

Instruction becomes available at the end of the Fetch stage

Operands at the end of Decode

Destination and ALU flags are updated at the end of Execute

```
...
sub     t1,t1,t2
addi    t2,t2,2
andi    t0,t0,1
slt     t2,t2,t0
```