Amdahl's Law

(a.k.a where to spend your efforts when improving performance)

$$t_{improved} = \frac{t_{affected}}{r_{speedup}} + t_{unaffected}$$

Example:

"Suppose a program runs in 100 seconds on a machine, where multiplies are executed 80% of the time. How much do we need to improve the speed of multiplication if we want the program to run 4 times faster?"

$25 = 80/r + 20, \quad r = 16x$

How about making it 5 times faster?

$20 = 80/r + 20, \quad r = ???$

Principle: Focus on making the most common case fast.

Amdahl's Law applies equally to H/W and S/W!

Example

Suppose we enhance a machine by making all floating-point instructions run **5** times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if only **50%** of the 10 seconds is spent executing floating-point instructions?

$6 = 5/5 + 5$ Relative Perf $= 10/6 = 1.67$ x

Marketing is looking for a benchmark to show off the new floating-point unit described above, and wants the overall benchmark to show at least a speedup of **3**. What percentage of the execution time would floating-point instructions have to be to account in order to yield our desired speedup on this benchmark?

$33.33 = p/5 + (100 - p) = 100 - 4p/5$ $p = 83.33$

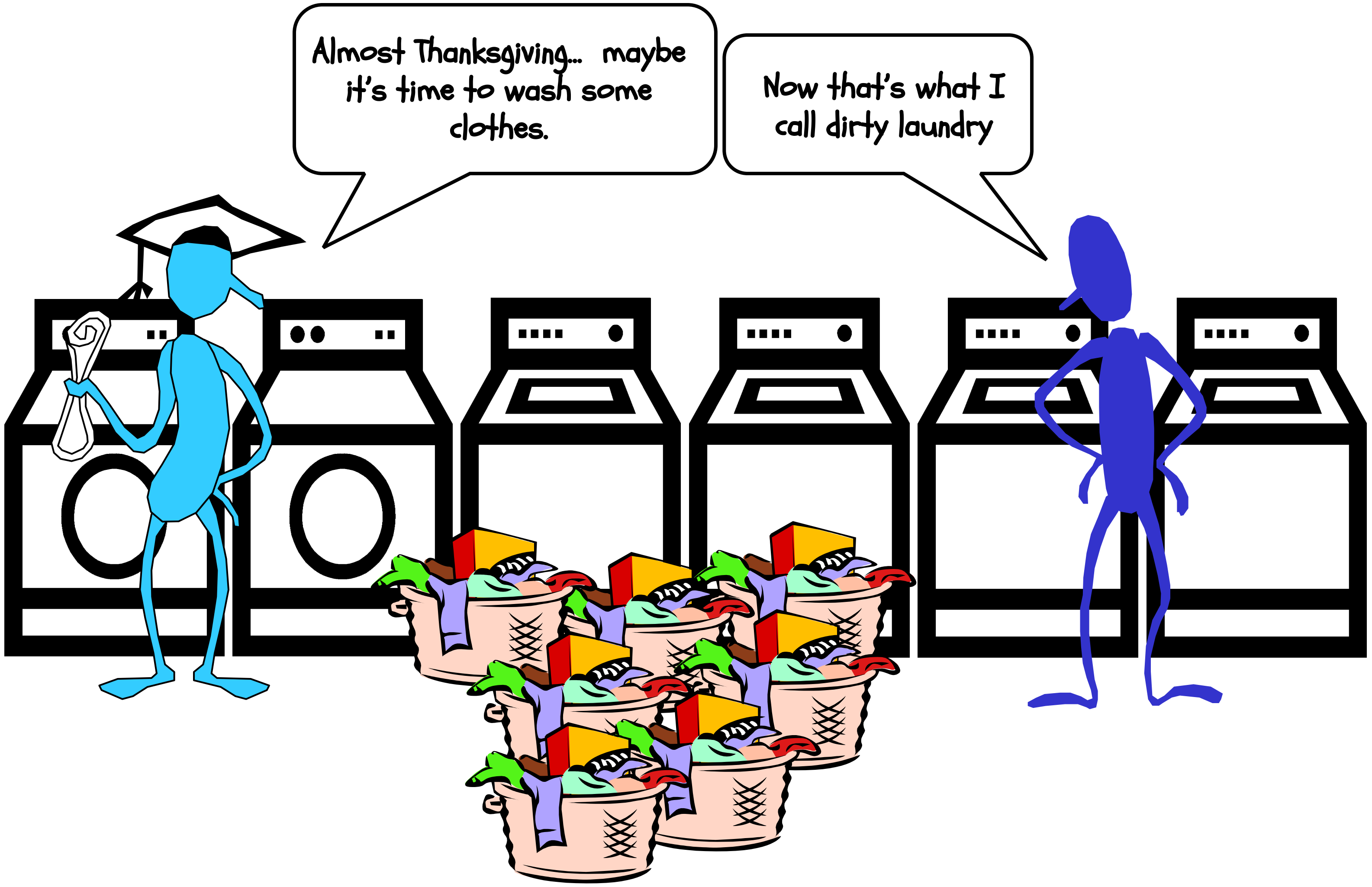
Remember

- **When performance is specific to a particular program**
  - Total execution time is a consistent summary of performance
- **For a given architecture performance comes from:**
  1) increases in clock rate (without adverse CPI affects)
  2) improvements in processor organization that lower CPI
  3) compiler enhancements that lower CPI and/or instruction count

- **Pitfall:** Advertized improvements in one aspect of a machine's performance affect the total performance

- You can't believe everything you read! So read carefully!

Pipelining


Almost Thanksgiving... maybe it's time to wash some clothes.
Now that's what I call dirty laundry

The Goal of Pipelining

- Recall our measure of processor performance

Millions of Instructions per Second

Frequency in Hz

$$\text{MIPS} = \frac{1}{10^6} \; \frac{\text{clocks / second}}{\text{clocks / instruction}}$$

CPI (Average Clocks Per Instruction)

- How can we turn up the clock rate?

Goal of Pipelining

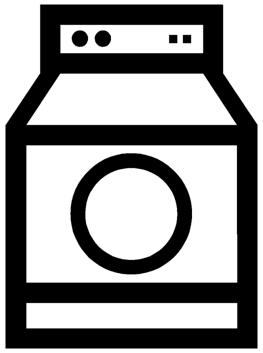
INPUT:
dirty laundry

OUTPUT:
4 more weeks

Device: Washer

Function: Fill, Agitate, Spin

Washer_{PD} = 30 mins



Device: Dryer

Function: Heat, Spin

Dryer_{PD} = 60 mins

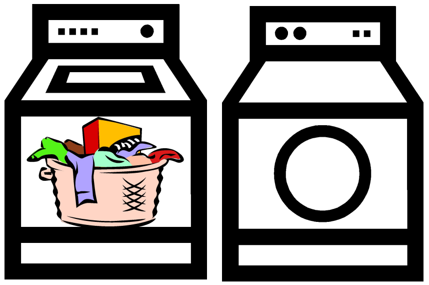
One Load at a Time

Everyone knows that the real reason that UNC students put off doing laundry so long is *not* because they procrastinate, are lazy, or even have better things to do.

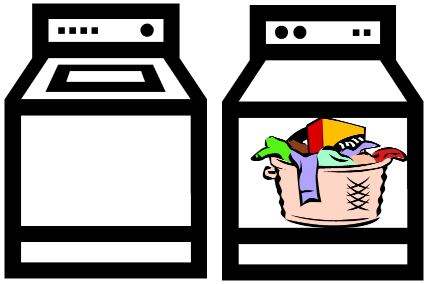
The fact is, doing laundry one load at a time is not smart.

(Sorry Mom, but you were wrong about this one!)

Step 1:



Step 2:



$$\text{Total} = \text{Washer}_{PD} + \text{Dryer}_{PD}$$

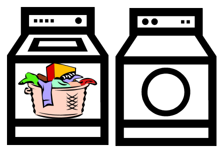
$$= \underline{\quad 90 \quad} \text{ mins}$$

Doing N Loads of Laundry

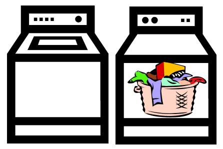
Here's how they do laundry at Duke, the "combinational" way.

(Actually, this is just an urban legend. No one at Duke actually does laundry. The butler's all arrive on Wednesday morning, pick up the dirty laundry and return it all pressed and starched by dinner)

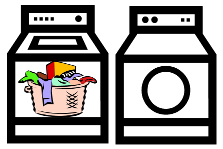
Step 1:



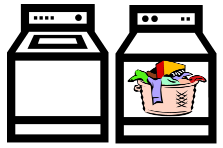
Step 2:



Step 3:



Step 4:



...

$$\text{Total} = N*(\text{Washer}_{PD} + \text{Dryer}_{PD})$$

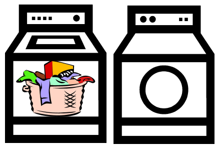
$$= \underline{\quad N*90 \quad} \text{ mins}$$

Doing N Loads… the UNC way

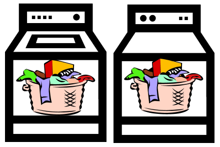
UNC students "pipeline" the laundry process.

That's why we wait!

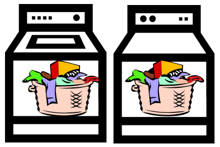
Step 1:



Step 2:



Step 3:



…

Actually, it's more like N*60 + 30 if we account for the startup transient correctly. When doing pipeline analysis, we're mostly interested in the "steady state" where we assume we have an infinite supply of inputs.

$$\text{Total} = N * \text{Max}(\text{Washer}_{PD}, \text{Dryer}_{PD})$$

$$= \underline{\quad N*60 \quad} \text{ mins}$$

Recall Our Performance Measures

Latency:

The delay from when an input is established until the output associated with that input becomes valid.

(Duke Laundry = ~~90~~ _________ mins)

(UNC Laundry = ~~120~~ _________ mins) ←

Assuming that the wash is started as soon as possible and waits (wet) in the washer until dryer is available.

Throughput:

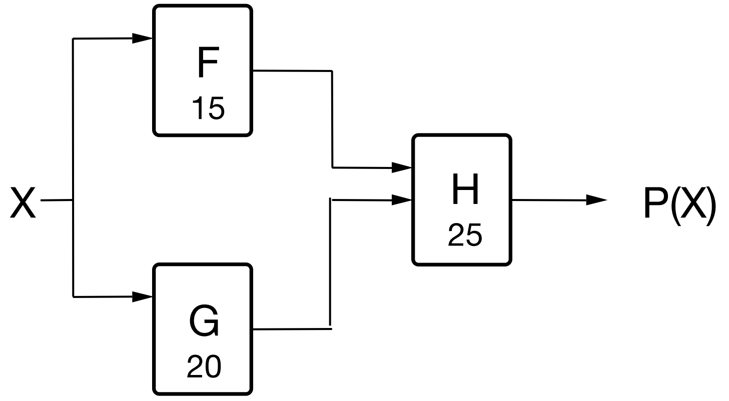
The rate of which inputs or outputs are processed.

(Duke Laundry = _____1/90_____ outputs/min)

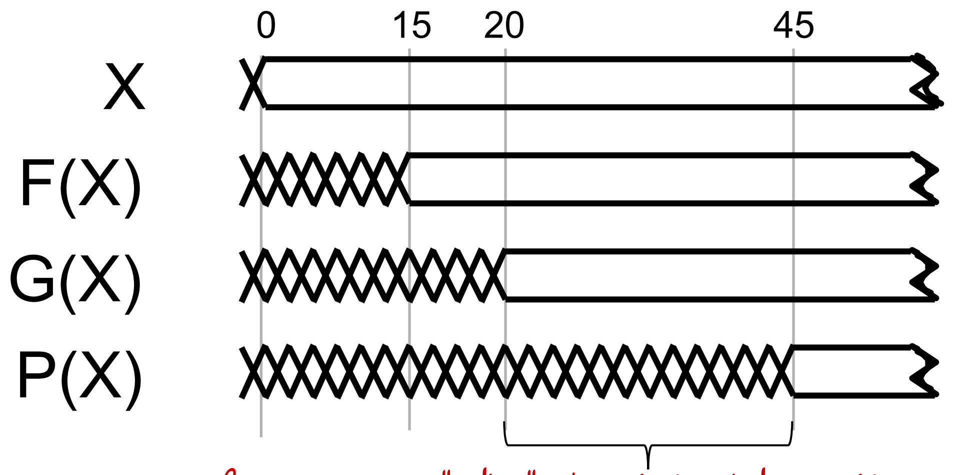
(UNC Laundry = _____1/60_____ outputs/min)

Even though we increase latency, it takes less time per load

Okay, Back to Circuits...



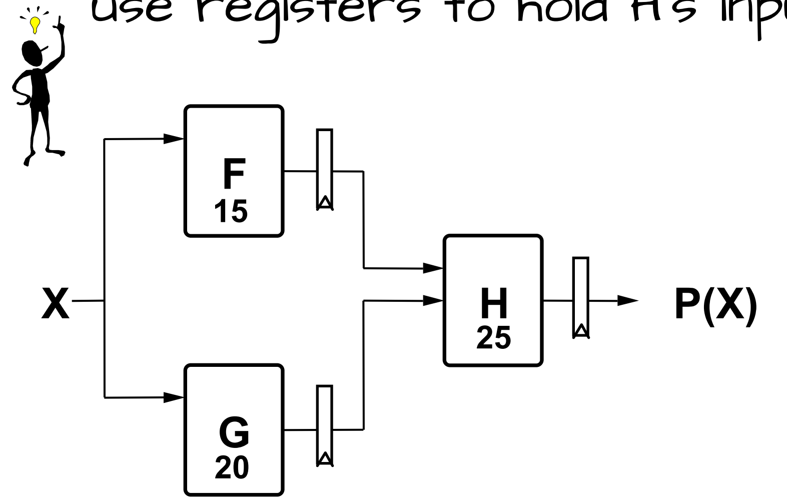
For combinational logic:
  latency = t_{PD},
  throughput = $1/t_{PD}$.
We can't get the answer faster, but are we making effective use of our hardware at all times?



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

Pipelined Circuits

use registers to hold H's input stable!



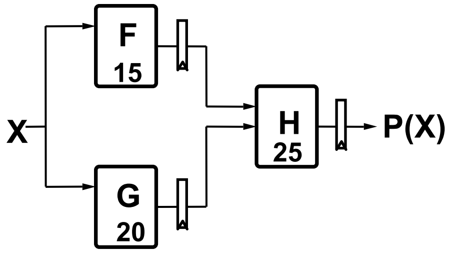
Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline* : if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

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

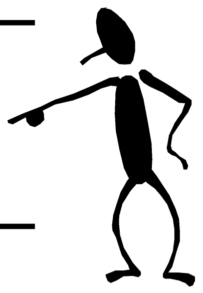
	latency	throughput
unpipelined	45	1/45
2-stage pipeline	50	1/25
	worse	better

Pipelining uses registers to improve the throughput of combinational circuits

Pipeline Diagrams



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



Clock cycle →

Pipeline stages ↓

	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			$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

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 CLOCK common to all registers *must* have a period sufficient to allow for the propagation delays of all combinational paths PLUS (input) register's t_{PD} PLUS (output) register's t_{SETUP}.

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.

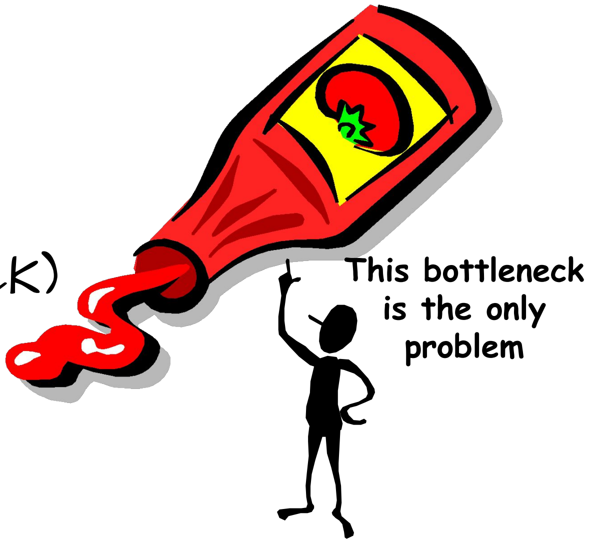
Pipelining Summary

Advantages:

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

Disadvantages:

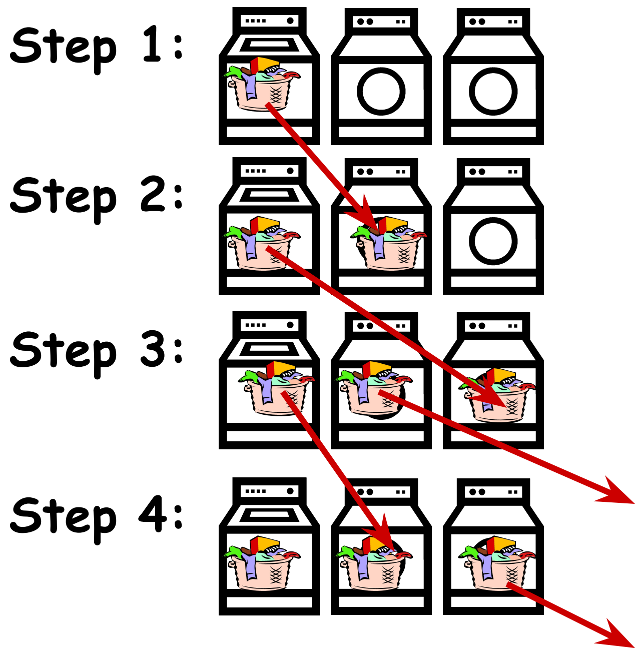
- Generally, increases latency
- Only as good as the *weakest* link (often called the pipeline's BOTTLENECK)



Isn't there a way around this "weak link" problem?

How UNC students REALLY do Laundry?

They work around the bottleneck.



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

Throughput = __1/30__ loads/min

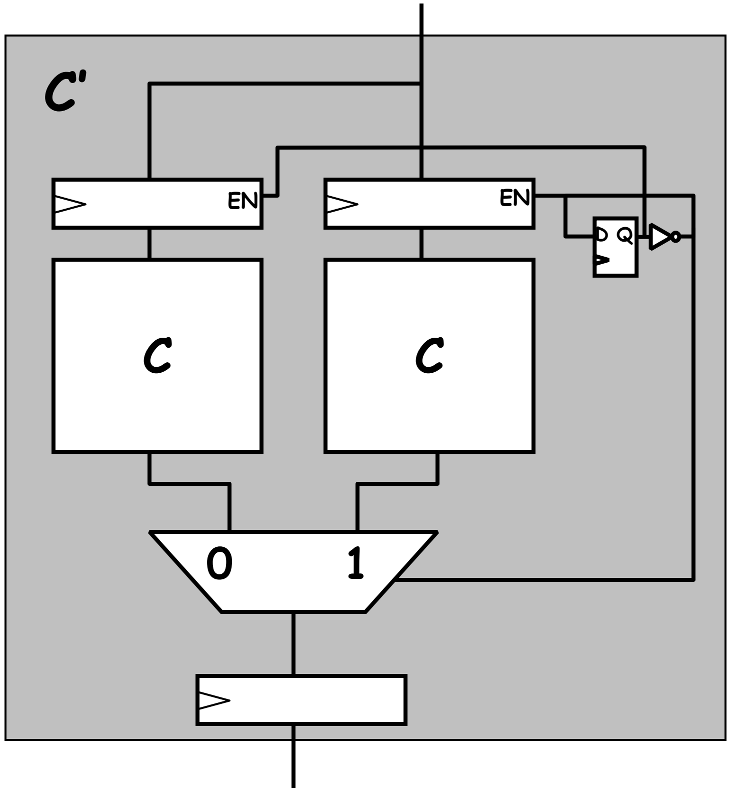
Latency = __90__ mins/load

Circuit 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.

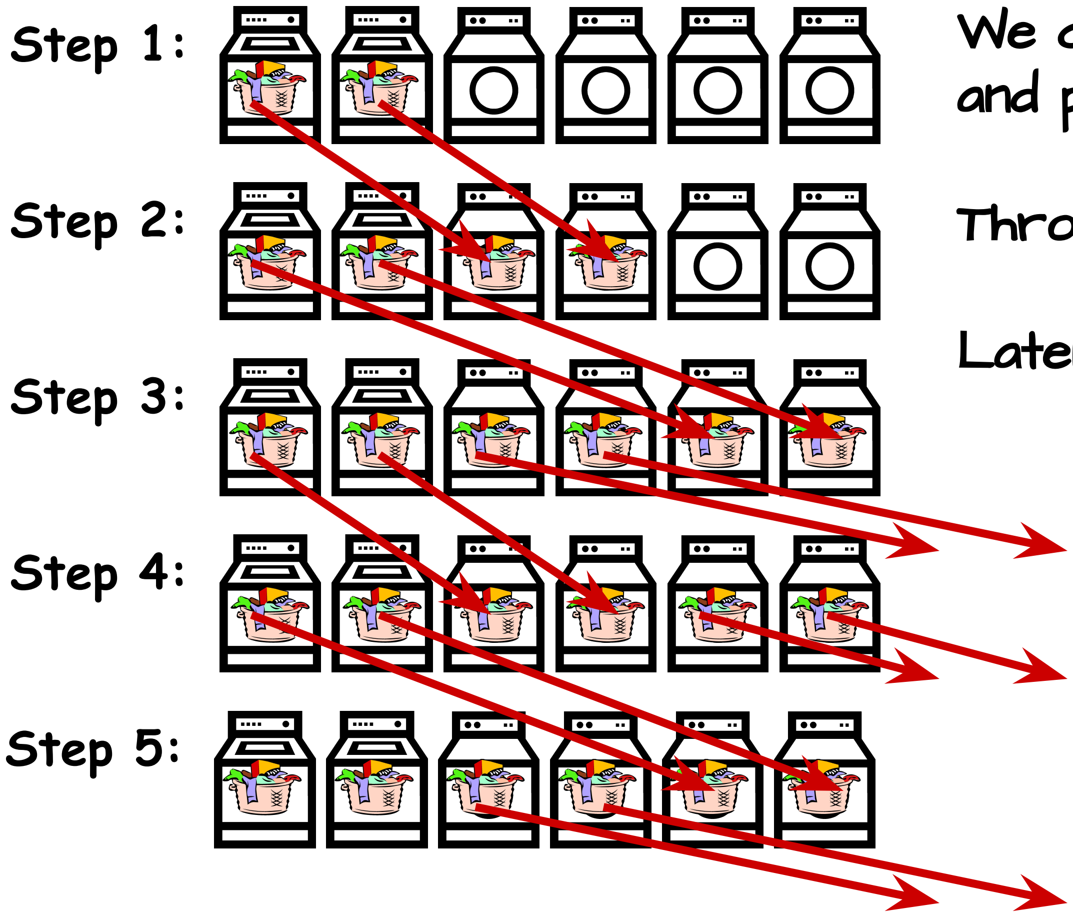
N-way interleaving is equivalent to how many pipeline stages? __**N**__

N-way
interleave



Latency = 2 clocks

Better Yet… Parallelism



We can combine interleaving and pipelining with parallelism.

Throughput = 1/15 load/min

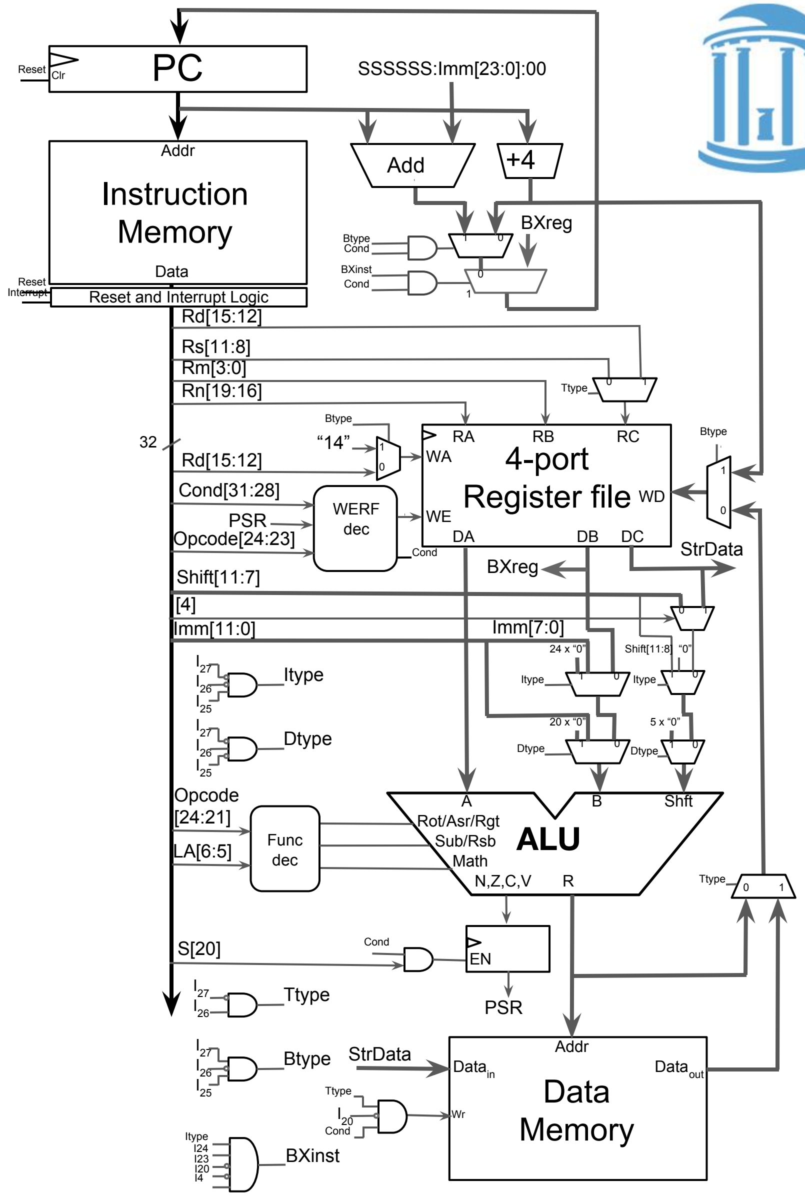
Latency = 90 min

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, PSR, 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:

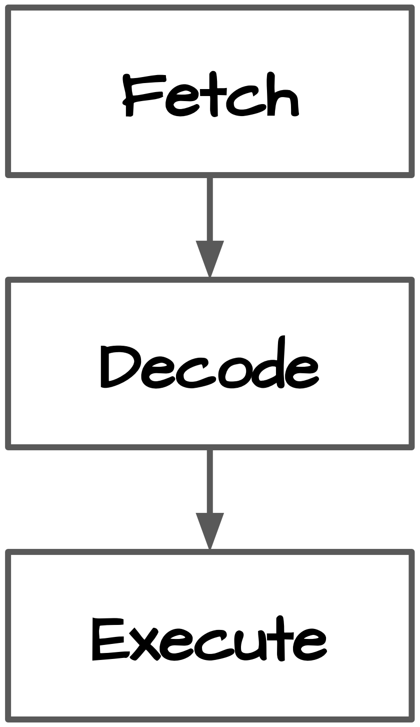
- Instruction memory access

Decode:

- Decode instructions
- Read register operands

Execute:

- ALU operation
- Write-back register



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      r1,r1,r2
add      r2,r2,#2
and      r0,r0,#1
cmp      r1,r2
```

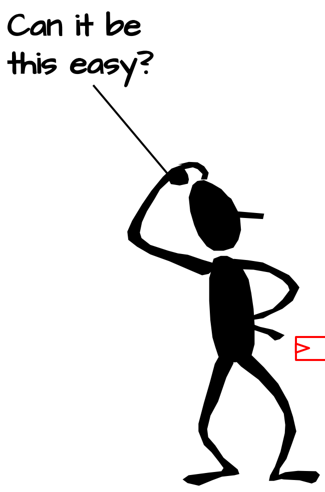
Time (in clock cycles) →

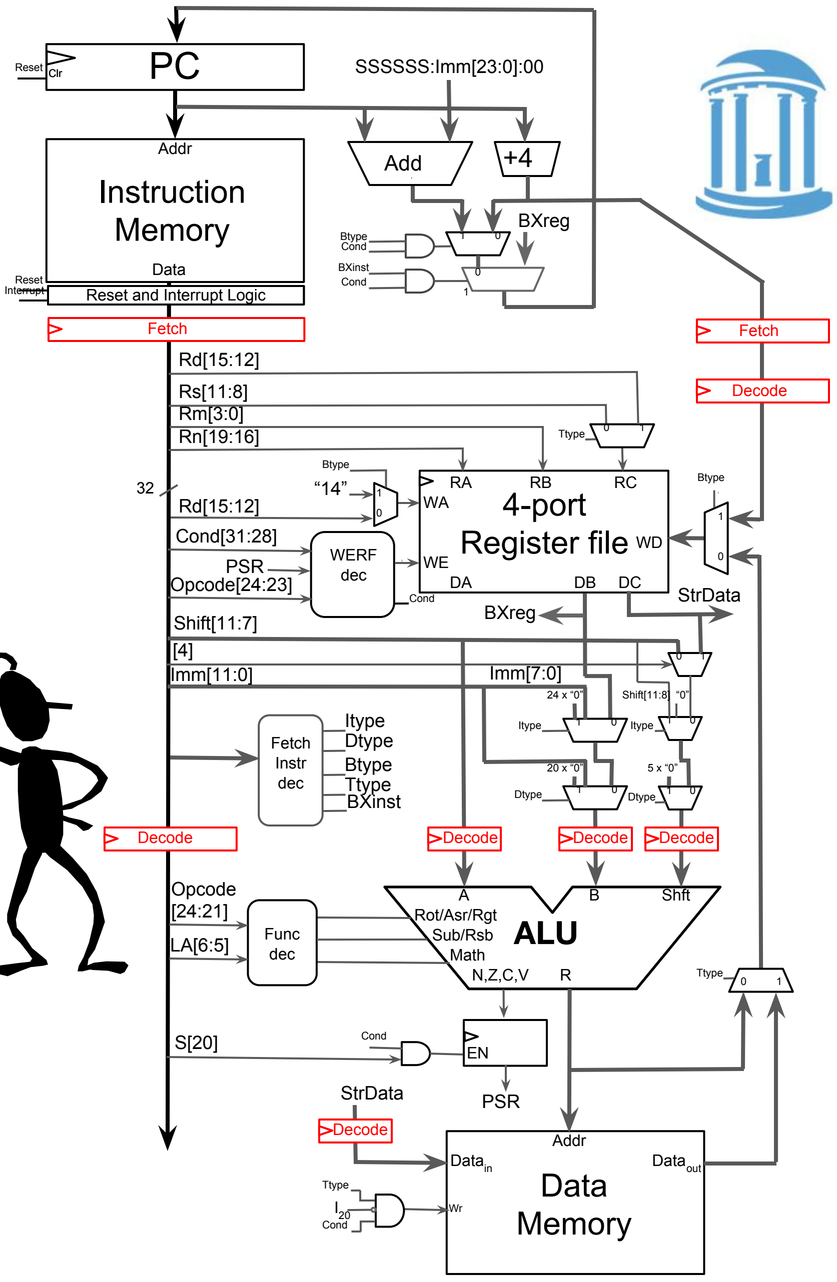
Pipeline ↓

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub r1,r1,r2	add r2,r2,#2	and r0,r0,#1	cmp r1,r2		
Decode		sub r1,r1,r2	add r2,r2,#2	and r0,r0,#1	cmp r1,r2	
Execute			sub r1,r1,r2	add r2,r2,#2	and r0,r0,#1	cmp r1,r2

Next Time

- Three pipeline registers on every datapath from the instruction memory's output to the register file's write data port.
- How much faster?




PC
Reset
Clr
SSSSSS:Imm[23:0]:00
Add
+4
Addr
Instruction Memory
Data
Reset
Interrupt
Reset and Interrupt Logic
Fetch
BXreg
Btype
Cond
BXinst
Cond
Decode
Rd[15:12]
Rs[11:8]
Rm[3:0]
Rn[19:16]
Ttype
32
Btype
"14"
Rd[15:12]
RA
RB
RC
WA
4-port Register file
WD
WE
Cond[31:28]
PSR
Opcode[24:23]
WERF dec
DA
DB
DC
Cond
StrData
BXreg
Shift[11:7]
[4]
Imm[11:0]
Imm[7:0]
24 x "0"
Shift[11:8]
"0"
Itype
Fetch Instr dec
Dtype
Btype
Ttype
BXinst
20 x "0"
5 x "0"
Opcode [24:21]
LA[6:5]
Func dec
A
B
Shft
Rot/Asr/Rgt
Sub/Rsb
Math
ALU
N,Z,C,V
R
S[20]
EN
PSR
StrData
Addr
Data in
Data out
Data Memory
Wr
I20