Pipelining and Real-world ISAs

Two topics today: ① Pipelining, ② Real-world ISAs

① Pipelining

Motivating example: laundry

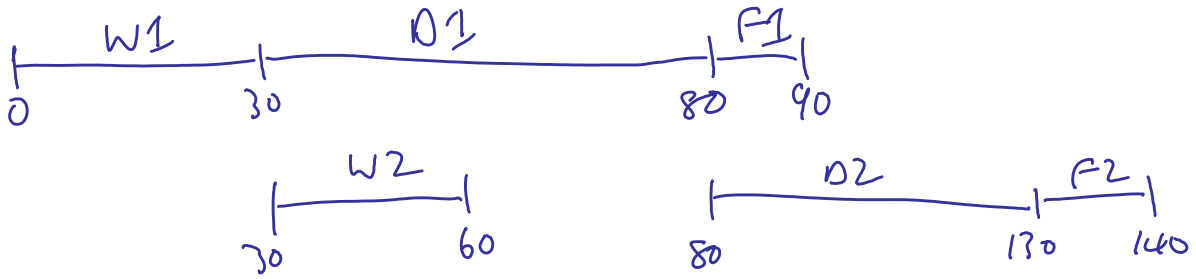
Suppose that your laundry process has 3 stages:

wash – 30 minutes
dry – 50 minutes
fold – 10 minutes.

total time for one load = 30+50+10 = 90 minutes.

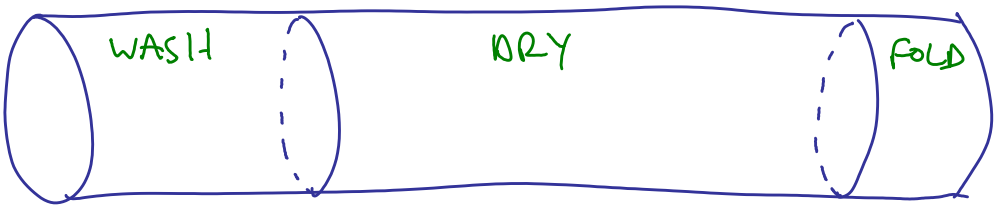
total time for two loads is <u>not</u> 2×90 = 180 minutes.

we can "pipeline" the process.
View as a timeline:



total time required is 140 minutes.

Can also view this using a pipeline metaphor:



any given stage can host only one load of laundry at a time.

Exercise: fill in the table:

number of loads	time required
1	90
2	140
100	
N	

Extension: Suppose we now have two dryers. How does this change the results?

(details as exercise)

... first dryer is no longer bottleneck. Washer is bottleneck, total time for N loads is $30N + 60$.

Processors do the same thing.

but note the Pentium 4 had 24 pipeline stages!

e.g. could have a 4-stage pipeline:

stage	description	
S1	fetch instruction	} each takes one cycle
S2	decode instruction	
S3	fetch operand	
S4	execute	

How many cycles do we need for the following program?

```
LOAD  X
SUBT  Y
ADD   W
STORE Z
```

complete table as exercise

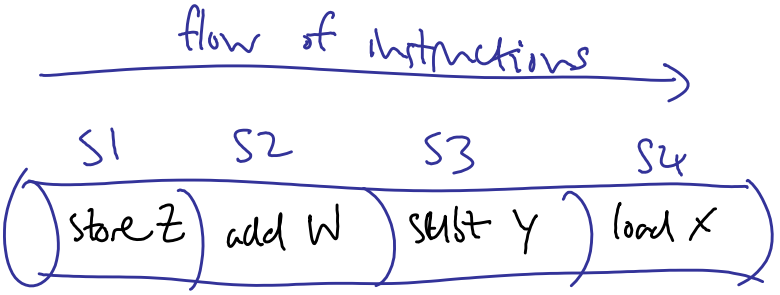
timeline view:

cycle:	1	2	3	4	5	6	7
load X	S1	S2	S3	S4			
subt Y							
add W							
store Z							

i.e. takes ___ cycles

fill in

Pipeline view:



How many cycles would be required for N instructions?

In the best case, we expect N+3 cycles but various problems can prevent us achieving this.

Remark: A non-pipelined CPU typically takes N cycles for N instructions.
The above 4-stage pipeline requires N+3 cyles for N instructions.
Is the pipelined CPU therefore slower?

Answer: No, because each stage is simpler so the pipelined CPU can have a much greater clock speed than the non-pipelined one.

Now back to problems that prevent optimal pipeline utilization:

Problem 1 Data dependencies cause delays

eg. same pipeline as above, executing the following program:

```
ADD   R1  X  Y
SUBT  R2  R1 Z
```

Cycle	1	2	3	4	5	6
ADD:	S1	S2	S3	S4		
SUBT:		S1	S2		S3	S4

can't fetch here. Operand is still being computed.

So this example requires an extra cycle.

Lesson: If input of one instruction depends on output of an earlier one, we might not be able to fully utilize the pipeline

Problem 2: Jumps cause delays

Example program:

0: Load X
1: Jump 6
2: Store Y
3: Halt
4: Halt
5: Halt
6: Store Z
7: Halt

all this work is wasted

Cycle	1	2	3	4	5	6	7	8	9	10
0:	S1	S2	S3	S4						
1:		S1	S2	S3	S4					
2:			S1	S2	S3					
3:				S1	S2					
4:					S1					
6:						S1	S2	S3	S4	
7:							S1	S2	S3	S4

i.e. required 10 cycles to execute 4 instructions.

Lessons:
- branch instructions cause poor pipeline utilization
- conditional branches are even more problematic
- modern processors do branch prediction or speculative execution to reduce this effect

simultaneously execute both branches and abandon the wrong one once its known

guess the most likely branch and start processing it

Real-world examples of ISAs

Historically, the two major design philosophies were

- RISC ("reduced instruction set computer")
  - small number of simple instructions that execute quickly
- CISC ("complex instruction set computer")
  - large number of complex instructions

Modern processors are typically hybrids of RISC and CISC, so the distinction has lost some of its importance.

(Historically, Intel was CISC; MIPS and ARM were RISC)

See the textbook for additional details on Intel and MIPS architectures.

The Java Virtual Machine provides another interesting example of an instruction set.

See the resources page for a minilab that investigates Java bytecode instructions.