Pipelining.

- Key idea:
  - Pipelining doesn't help latency of a single task, it helps throughput of the entire workload.
  - Multiple tasks operating simultaneously using different resources.
  - Possible delays:
    - Pipeline rate limited by slowest pipeline stage
    - Stall for dependencies

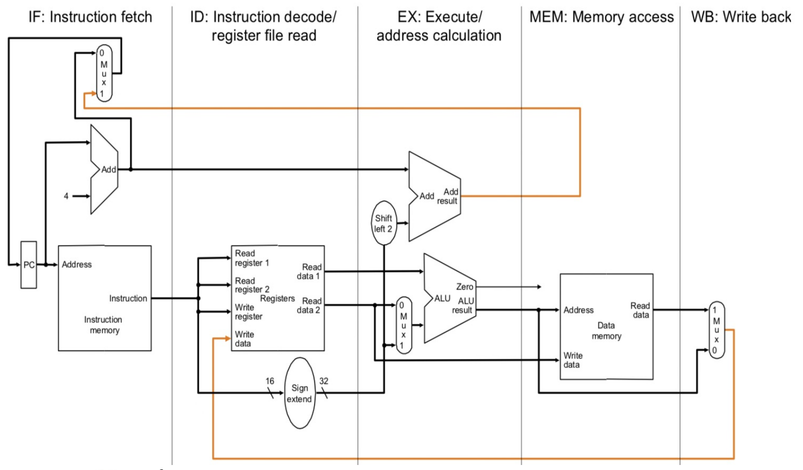
- 5 Execution Stages
  - IF: Instruction Fetch
  - ID: Instruction Decode and Register Read
  - EX: Execute an operation or calculate an address.
  - MEM: Access an operand in data memory. (lw/sw)
  - WB: Write back the result into a register. (R/I format)

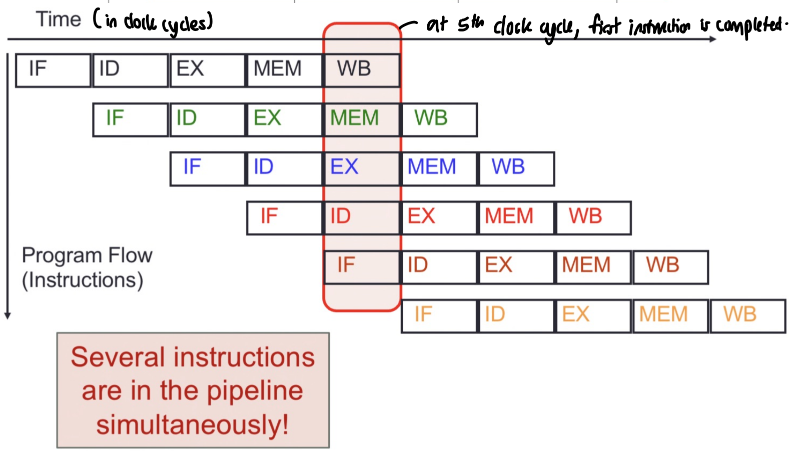
Each execution stage takes 1 clock cycle.

General flow of data is from one stage to the next. (except for update of PC and write back of register file).

"non-pipelined"
↳ All stages → 1 clock cycle.

Hardware elements:





Time (in clock cycles)
at 5th clock cycle, first instruction is completed.
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
Program Flow (Instructions)
Several instructions are in the pipeline simultaneously!

Datapath for pipelining

same instruction

Single Cycle: update all state elements (PC, register file, data memory) at the end of a clock cycle.

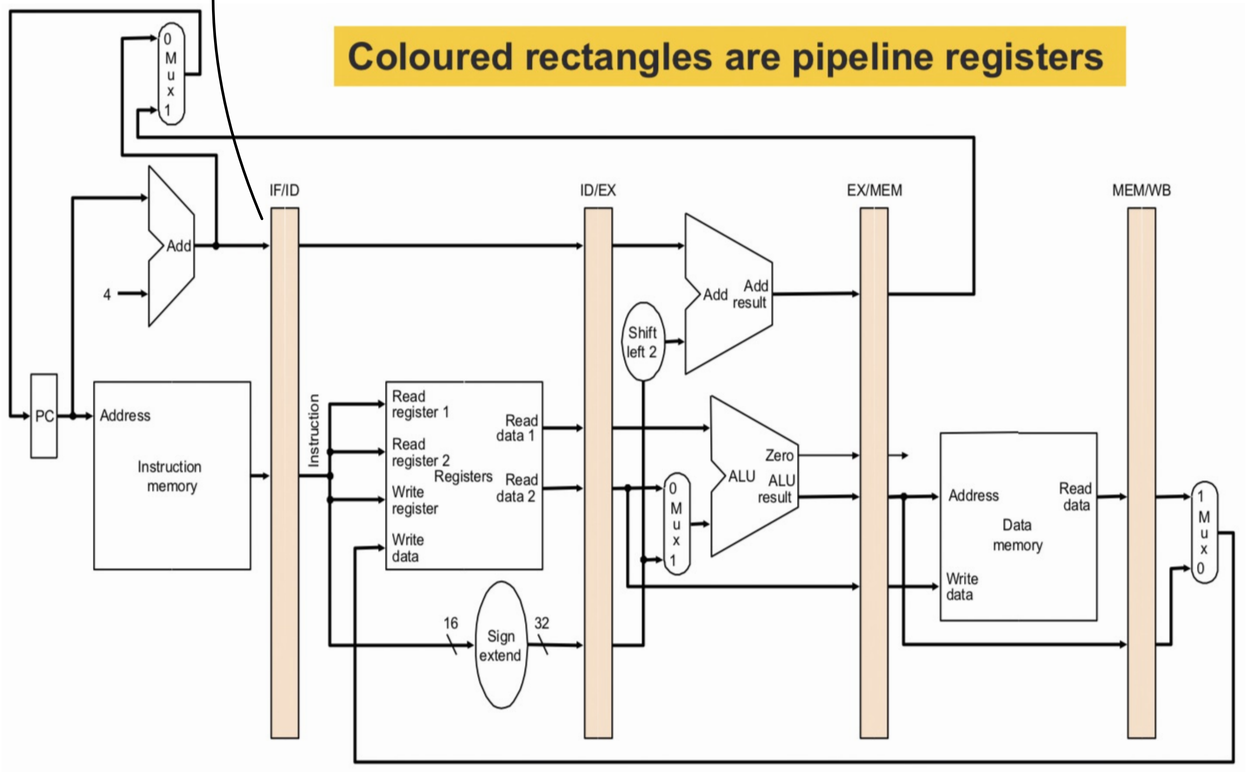
Pipelined Implementation: One cycle per pipeline stage.
Data required for each stage needs to be stored separately.

Data used by subsequent Instructions:
- Stored in programmer-visible state elements, eg. PC, register file, memory.

Data used by same instruction in later pipeline stages:
- Additional registers in datapath called pipeline registers
- IF/ID: register btw IF and ID.
- ID/EX: register btw ID and EX.
- EX/MEM: register btw EX and MEM.
- MEM/WB: register btw MEM and WB.

nothing to store at the end of WB
∴ no register at the end of WB.

"Collection" of registers.
→ To store data.



IF stage:

- At the end of a cycle, **IF/ID** receives (stores):
  - Instruction read from InstructionMemory[PC]
  - PC + 4
- PC + 4
  - Also connected to one of the MUX's inputs (another coming later)

ID stage:

At the beginning of a cycle **IF/ID register supplies:**	**At the end of a cycle** **ID/EX receives:**
From Instr. { ❖ Register numbers for reading two registers ❖ 16-bit offset to be sign-extended to 32-bit ❖ **PC + 4**	❖ Data values read from register file ❖ 32-bit immediate value ❖ **PC + 4**

Ex stage:

At the beginning of a cycle **ID/EX register supplies:**	**At the end of a cycle** **EX/MEM receives:**
❖ Data values read from register file ❖ 32-bit immediate value ❖ **PC + 4**	❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ **isZero?** signal ❖ Data Read 2 from register file

MEM Stage:

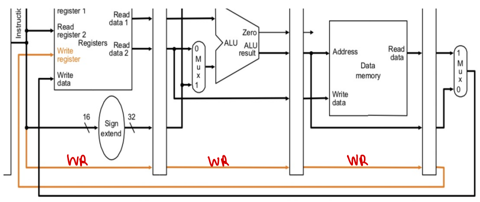
At the beginning of a cycle **EX/MEM register supplies:**	**At the end of a cycle** **MEM/WB receives:**
❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ **isZero?** signal ❖ Data Read 2 from register file	❖ ALU result ❖ Memory read data

WB Stage:

At the beginning of a cycle **MEM/WB register supplies:**	**At the end of a cycle**
❖ ALU result ❖ Memory read data	❖ Result is written back to register file (if applicable) ❖ **There is a bug here.......**

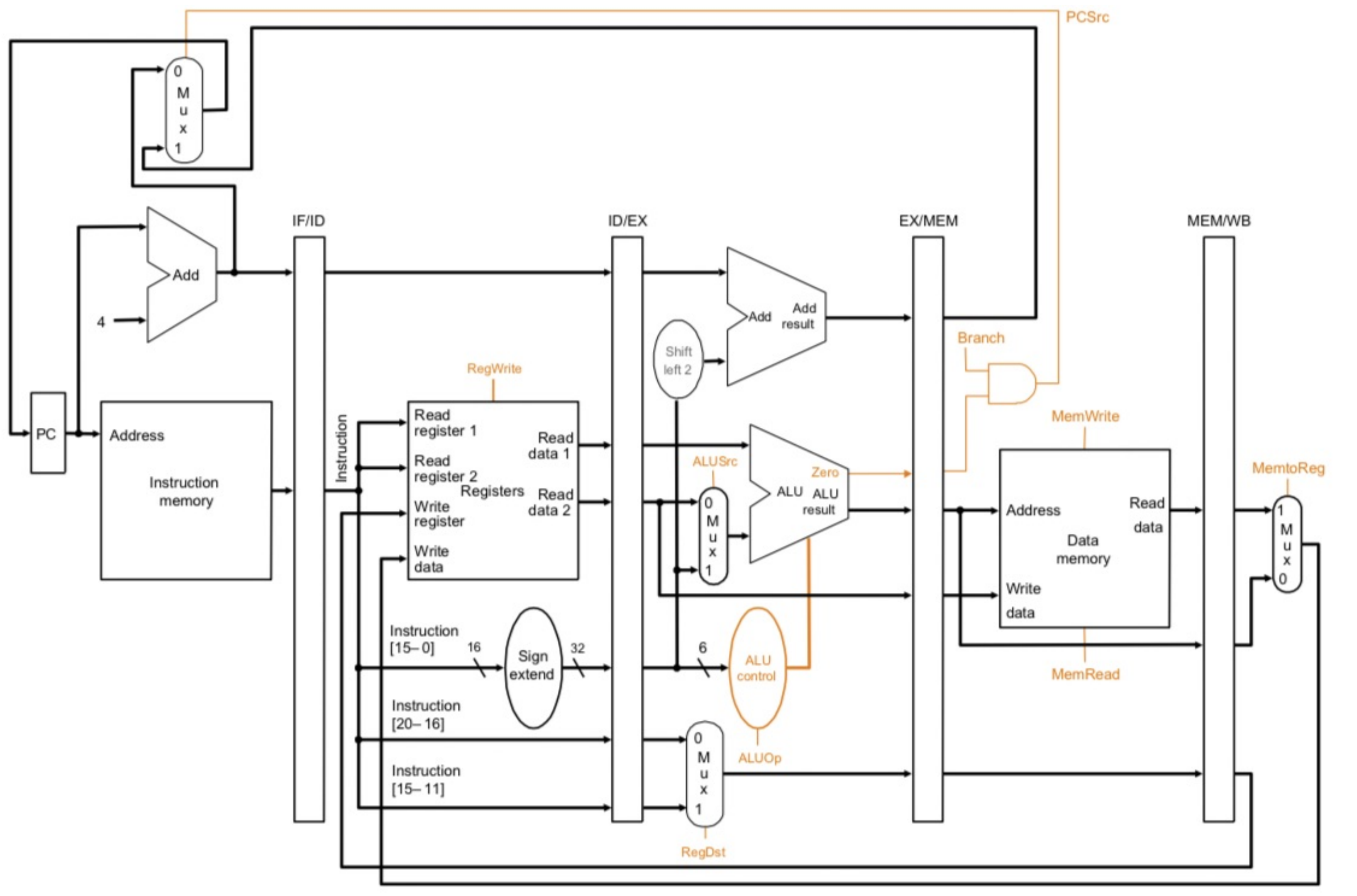
↳ Write Register in RegFile might be different.

∴ need to carry Write Register Number all the way.



Control Path for Pipelining.

- Same control signals as single-cycle datapath.
- Each control signal belongs to a particular pipeline stage.

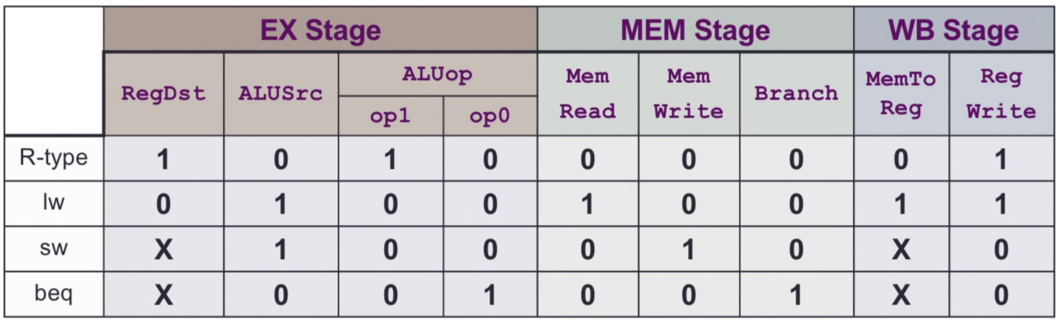


Decode Stage:
- get 6 bit op-code to generate control signal

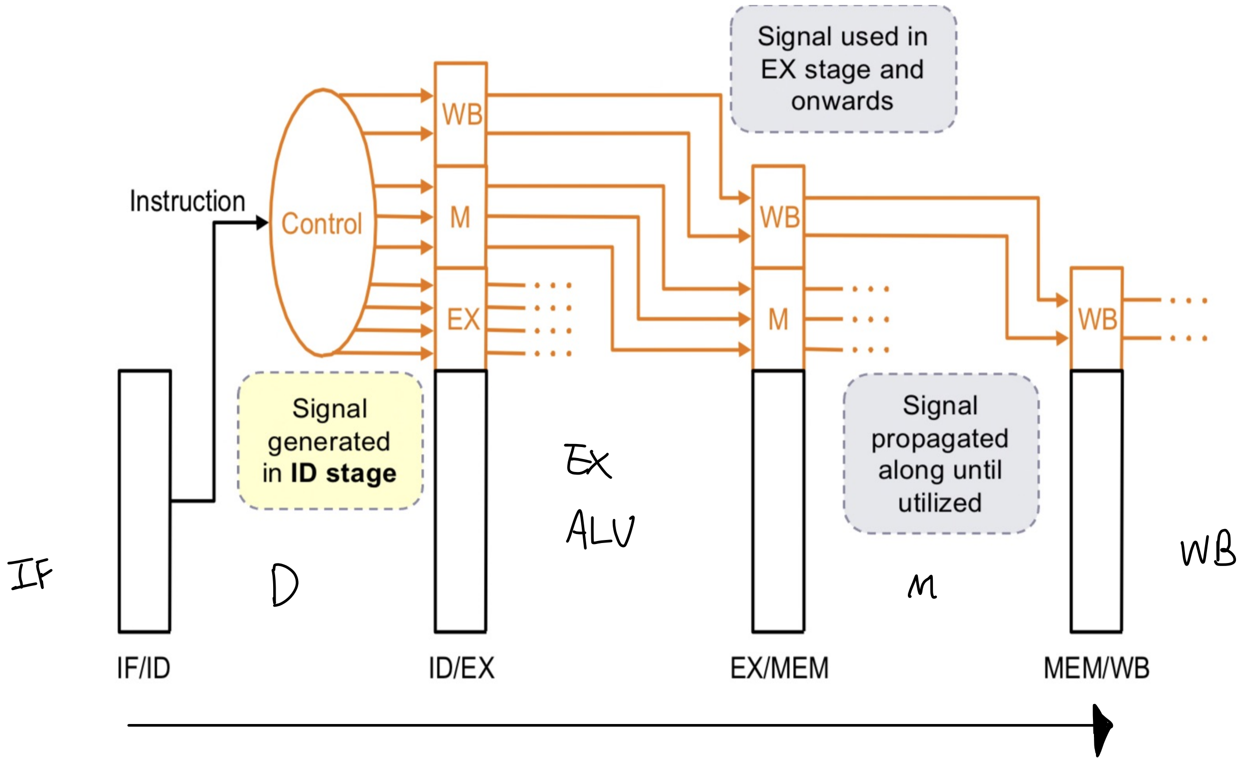
EX:
- ALUSrc
- ALUOp
- RegDst

MEM:
- MemRead
- MemWrite
- Branch.

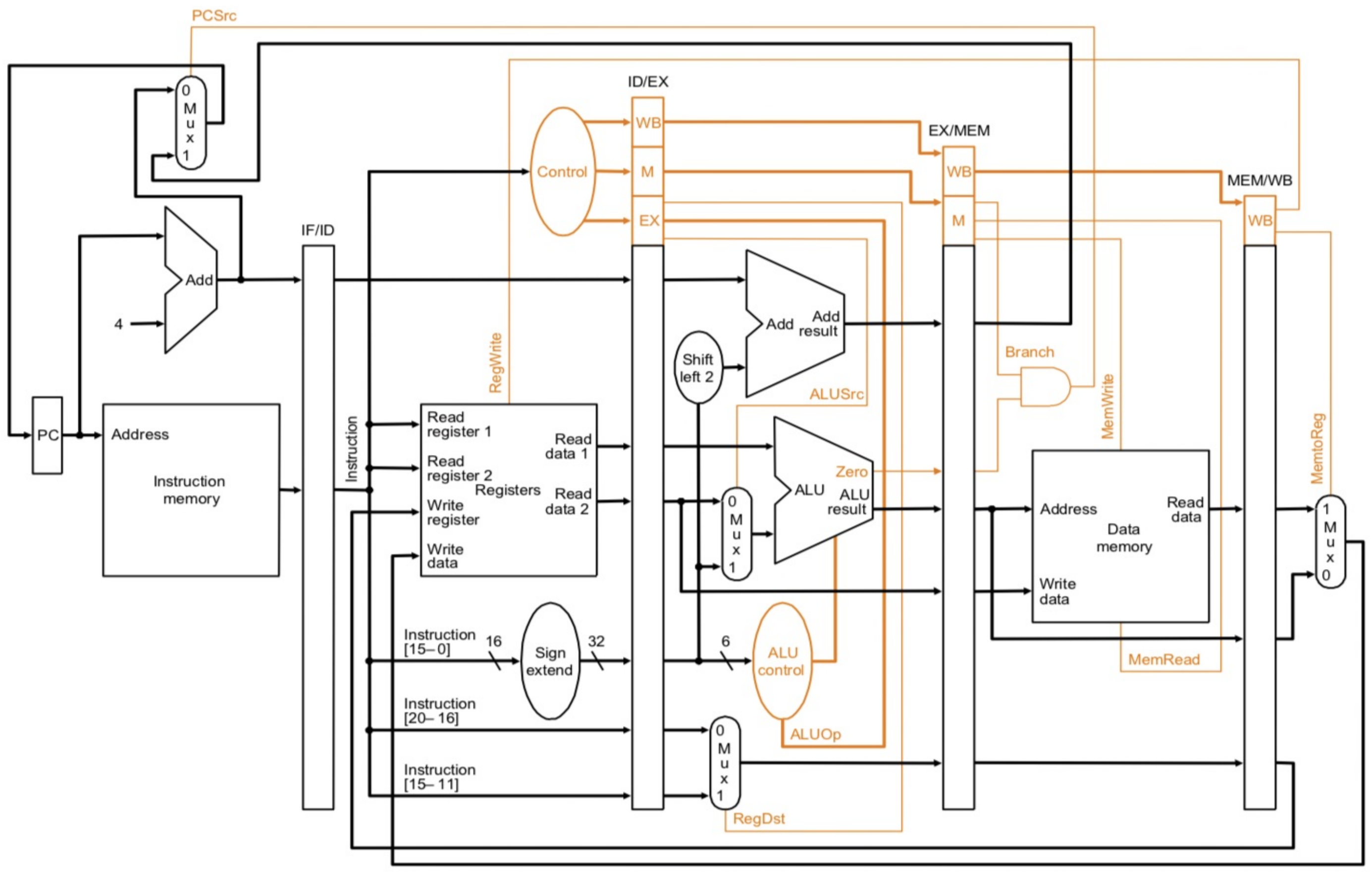
WB:
- MemToReg.
- RegWrite.



	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0

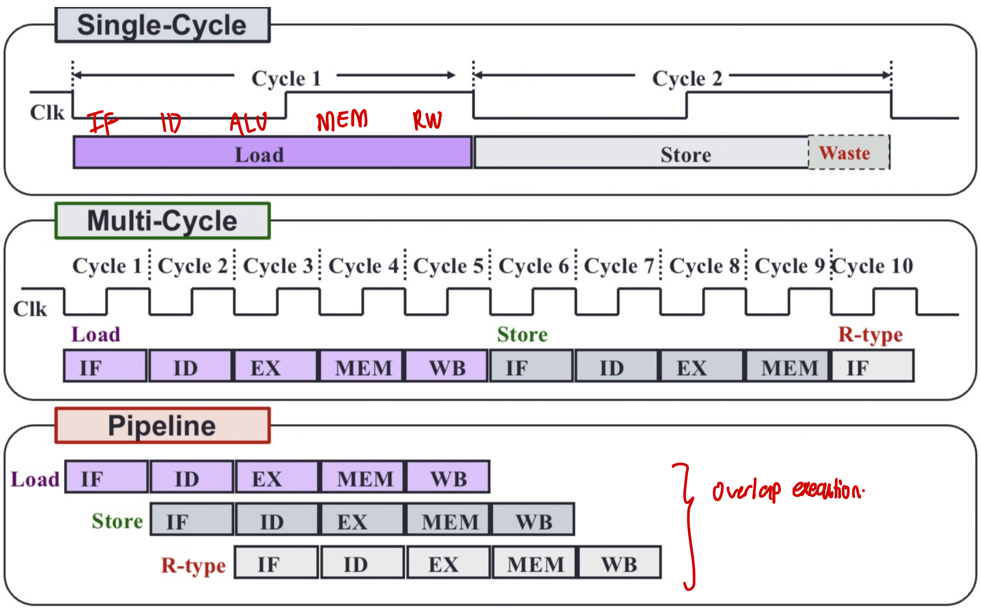


Signal used in EX stage and onwards
Instruction
Control
WB
M
EX
WB
M
WB
Signal generated in **ID stage**
Signal propagated along until utilized
IF
D
EX
ALU
M
WB
IF/ID
ID/EX
EX/MEM
MEM/WB



PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
Add
4
Add
Add result
Shift left 2
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALUSrc
ALU
Zero
ALU result
Branch
MemWrite
Address
Data memory
Write data
Read data
MemtoReg
MemRead
Instruction [15–0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst

Different Implementations



one big clock cycle for all the stages.

one small clock cycle for each stage.

overlap execution.

Single-Cycle Processor

- Cycle time: $CT_{seq} = \max(\sum_{k=1}^{N} \underset{\uparrow}{T_k})$ max of instrs. IF/D/A/M/W
  - T_k = Time for operation in stage k
  - N = Number of stages
- Execution Time for I instructions:
  - $Time_{seq} = I \times CT_{seq}$

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU (eg: add)	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

Assume 100 instructions.

Cycle time = 8ns (max cycle time of all instrs.)

Time to execute 100 instructions = 100×8ns = 800ns

Multi-Cycle Processor

- Cycle time: $CT_{multi} = \max(T_k)$ IF/D/A/M/W max of the stages.
- Execution Time for I instructions:
  - $Time_{multi} = I \times Average\ CPI \times CT_{multi}$
  - Average CPI needed as each instruction takes different number of cycles (diff. no. of stages) via statistics.

Cycle time = 2ns

Given average CPI = 4.6

Time to execute 100 instructions = 100 × 4.6 × 2ns = 920ns

Pipelining Processor

max of stages. IF/D/A/M/W i.e. accessing the pipeline register.

- Cycle time: $CT_{pipeline} = \max(T_k) + T_d$ overhead
  - T_d = Overhead for pipelining, e.g. pipeline register
- Cycles needed for I instructions:
  - $I + N - 1$ (need $N - 1$ cycles to fill up the pipeline)
- Execution Time for I instructions:
  - $Time_{pipeline} = (I + N - 1) \times CT_{pipeline}$

N is number of stages.

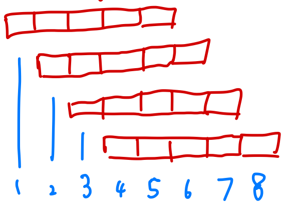
Assume pipeline register latency = 0.5ns

Cycle time = 2ns + 0.5ns = 2.5ns

Time to execute 100 instructions = (100+5−1) × 2.5ns = 260ns.

eg. 4 instr., 5 stages ⇒ 4+(5−1) = 8 cycles.

add



Ideal Speedup.

- Assumptions for ideal case:
  - Every stage takes the same amount of time → $\sum_{k=1}^{N} T_k = N \times T_1$
  - No pipeline overhead → $T_d = 0$
  - $I >> N$ (Number of instructions is much larger than number of stages)

- $Speedup_{pipeline} = \dfrac{Time_{seq}}{Time_{pipeline}}$ (w/o pipeline) (with pipeline).

$$= \frac{I \times \sum_{k=1}^{N} T_k}{(I+N-1) \times (\max(T_k) + \cancel{T_d})} = \frac{I \times N \times T_1}{(I+N-1) \times T_1}$$

0

all stages taken same amt. of time.

$$\approx \frac{\cancel{I} \times N \times \cancel{T_1}}{\cancel{I} \times \cancel{T_1}}$$

since $I >> N$

$$= \boldsymbol{N}$$

Conclusion:
Pipeline processor can gain **N** times speedup, where **N** is the number of pipeline stages.

Review Question

- Given this code:

```
add $t0, $s0, $s1
sub $t1, $s0, $s1
sll $t2, $s0, 2
srl $t3, $s1, 2
```

a) How many cycles will it take to execute the code on a single-cycle datapath?

b) How long will it take to execute the code on a single-cycle datapath, assuming a 100 MHz clock? 1 Hz = 1 cycle per sec.

c) How many cycles will it take to execute the code on a 5-stage MIPS pipeline? $4 + (5-1) = 8$.

d) How long will it take to execute the code on a 5-stage MIPS pipeline, assuming a 500 MHz clock?

a) 4 cycles

b) $4/(100 \times 10^6) = 40$ ns

c) $4 + 4 = 8$ cycles

d) $8/(500 \times 10^6) = 16$ ns

* Technique: (ideal)

no. of stages + no. of instructions − 1

$$\frac{10 \times 8}{10} \times \frac{1}{500 \times 10^6} = \frac{16}{10^9} = 16\text{ ns}.$$