Pipelining and Branch Prediction

CPU Design 101

Hai Cao

April 20, 2023

Dept. of Electronics Engineering

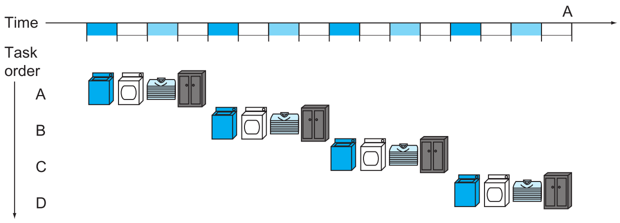
Table of contents

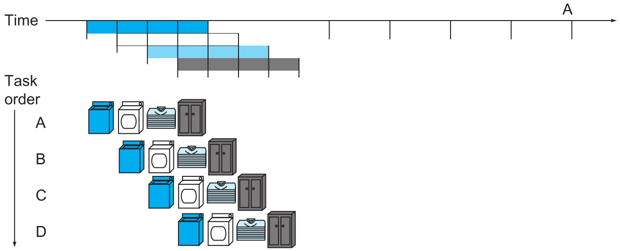
Objectives

- Understand the pipeline technique
- Acknowledge and fix hazards
- Understand basic branch prediction technique

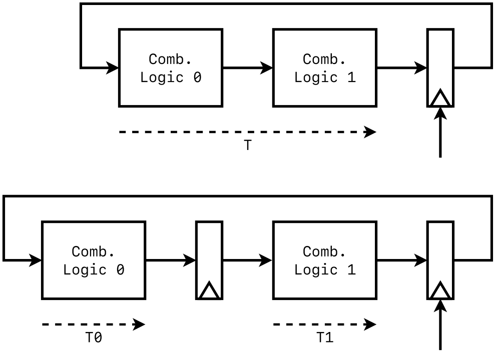
Pipelining

Overview





Overview



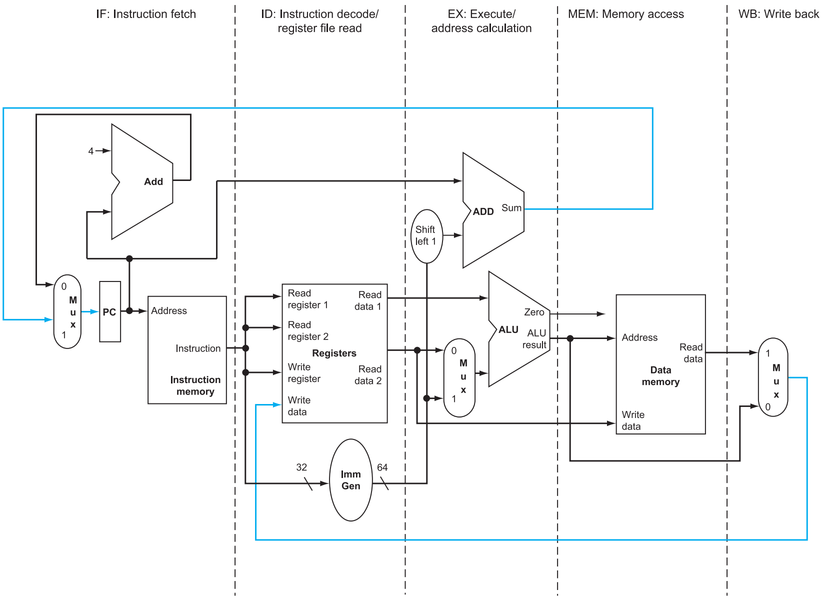
$$f(clk)_{unpipelined} \leqslant \frac{1}{T}$$

$$f(clk)_{pipelined} \leqslant \min\left(\frac{1}{T_0}, \frac{1}{T_1}\right)$$

Pipelining in Real-life

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
Basically, RISC-V instructions take 5 steps

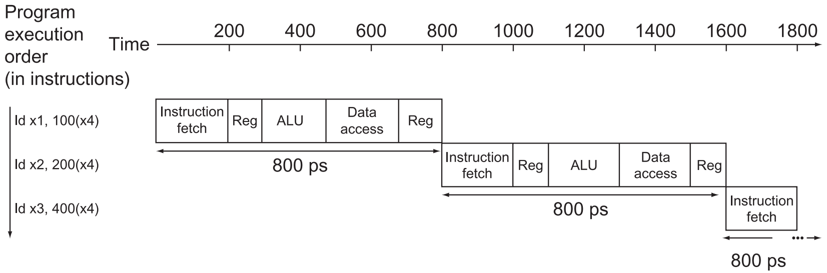
IF	**Fetch** instruction from memory
ID	**Decode** instruction and read Regfile
EX	**Execute** the operation or compute an address
MEM	**Memory** is accessed for loading or storing data (if neccessary)
WB	**Write Back** data to Regfile (if neccessary)

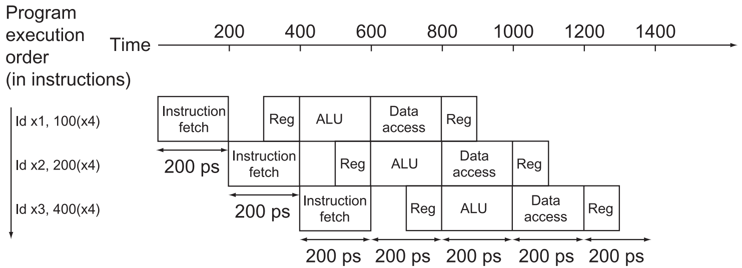

IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
4
Add
ADD
Sum
Shift left 1
0
M
u
x
1
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
ALU
Zero
ALU result
0
M
u
x
1
Address
Read data
Data memory
Write data
1
M
u
x
0
32
Imm Gen
64

Pipelining in Real-life

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Pipelining in Real-life





Speed-up

Given:

m	the number of instructions
n	the number of pipelined stages
t	the required time of each stage to complete

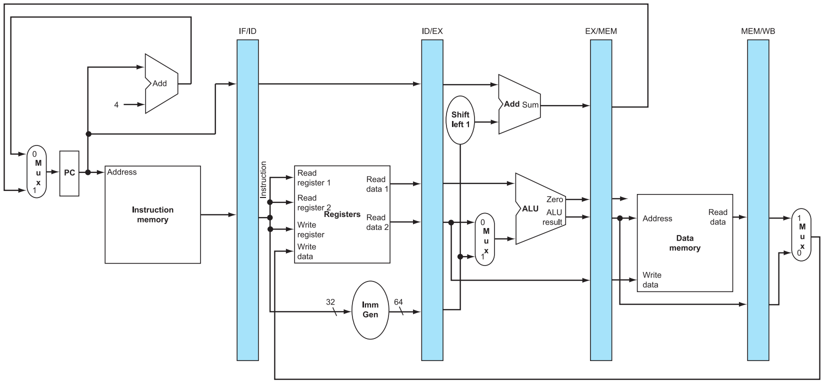
The completed time

without	pipelining mnt
with	pipelining $(m+n-1)t$

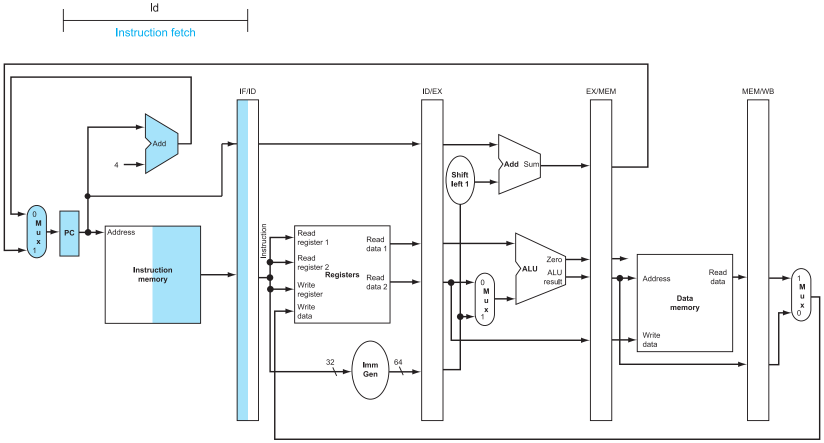
$$S(n) = \frac{mnt}{(m+n-1)t} = \frac{mn}{m+n-1}$$

$$\lim_{m\to\infty} S(n) = n$$

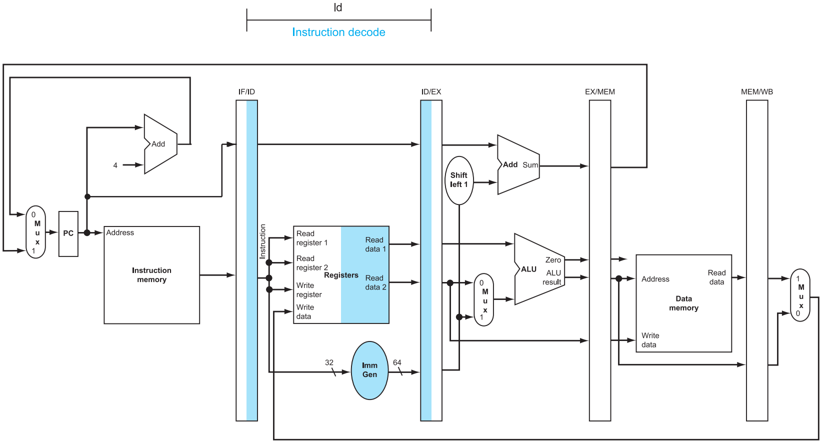
Seperating Stages



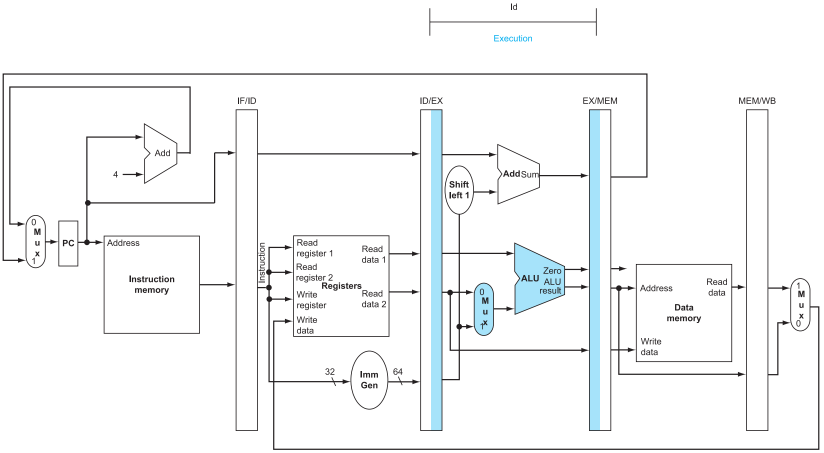
LOAD in IF Stage



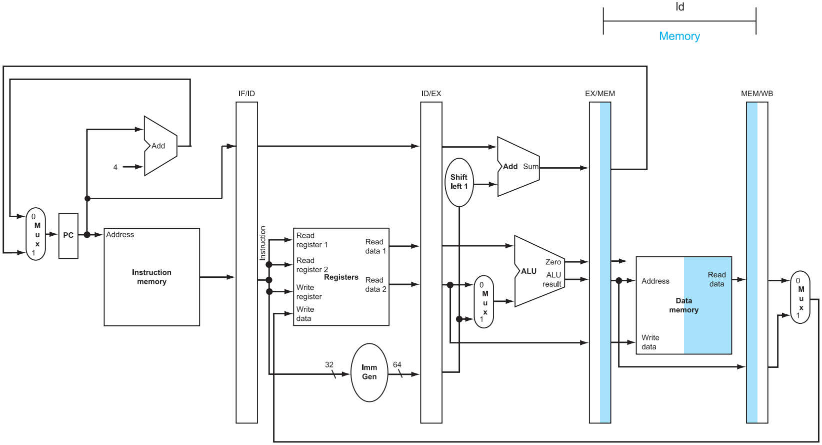
LOAD in ID Stage



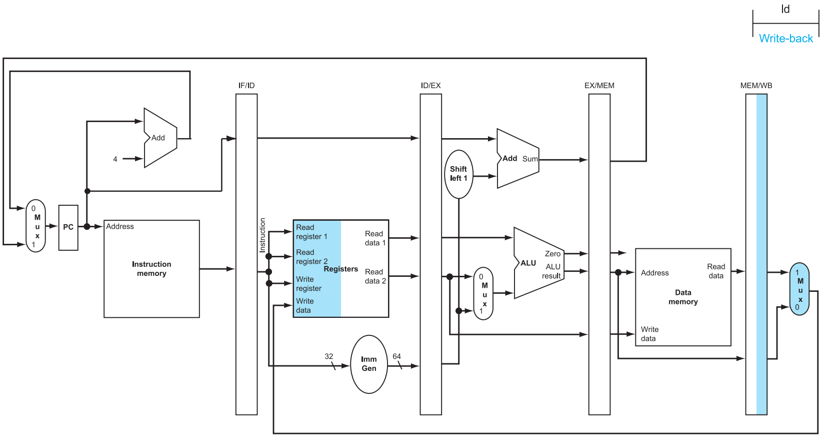
LOAD in EX Stage



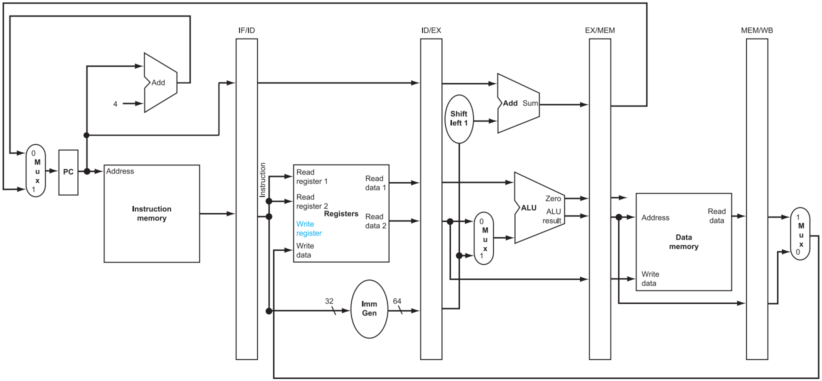
LOAD in MEM Stage



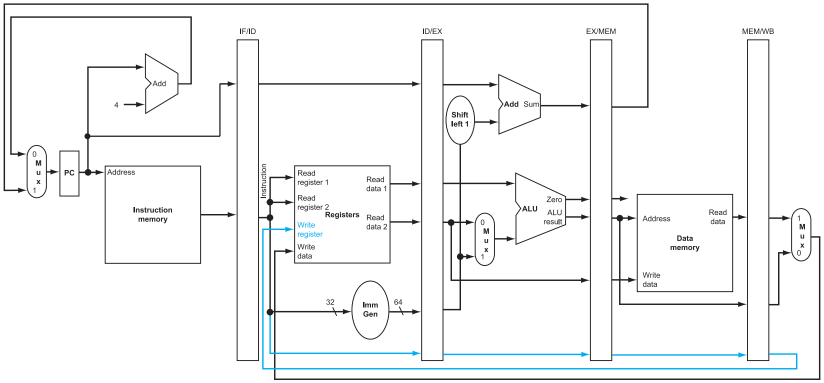
LOAD in WB Stage



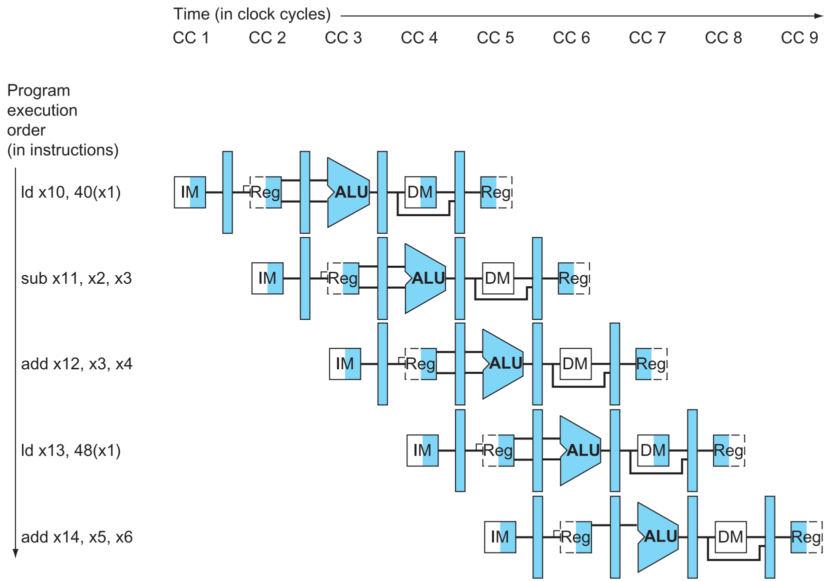
How to Wire RegWren



How to Wire RegWren

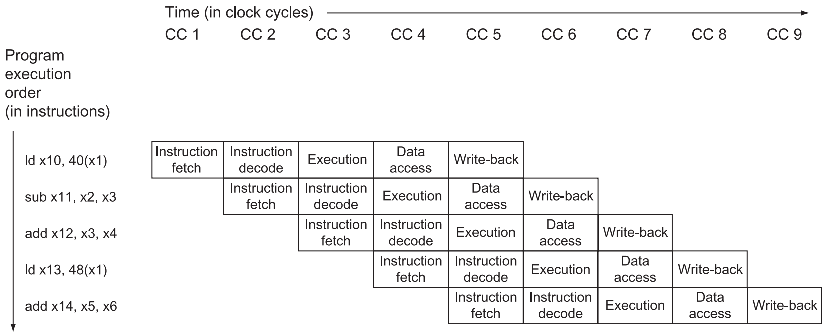


Pipelining



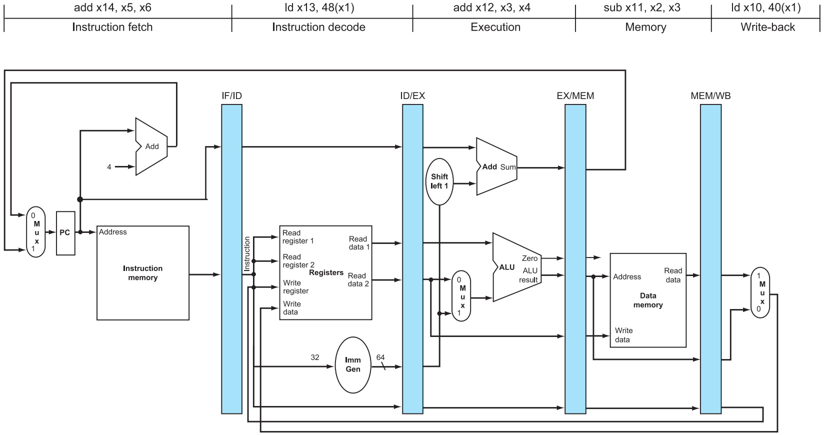
Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
ld x10, 40(x1)
sub x11, x2, x3
add x12, x3, x4
ld x13, 48(x1)
add x14, x5, x6
IM
Reg
ALU
DM
Reg

Pipelining

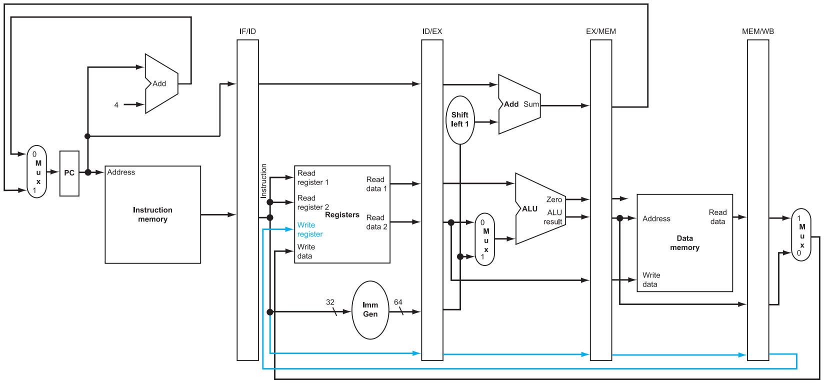


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
ld x10, 40(x1)
Instruction fetch
Instruction decode
Execution
Data access
Write-back
sub x11, x2, x3
Instruction fetch
Instruction decode
Execution
Data access
Write-back
add x12, x3, x4
Instruction fetch
Instruction decode
Execution
Data access
Write-back
ld x13, 48(x1)
Instruction fetch
Instruction decode
Execution
Data access
Write-back
add x14, x5, x6
Instruction fetch
Instruction decode
Execution
Data access
Write-back

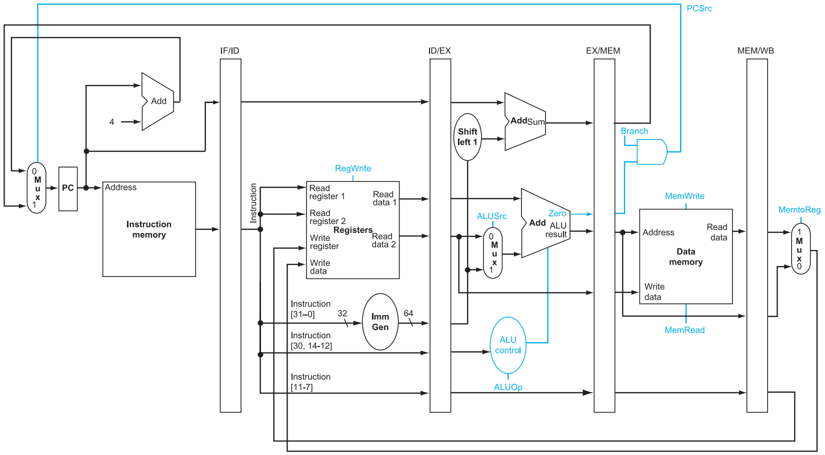
Pipelining


add x14, x5, x6
Instruction fetch
ld x13, 48(x1)
Instruction decode
add x12, x3, x4
Execution
sub x11, x2, x3
Memory
ld x10, 40(x1)
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add Sum
0 M u x 1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1 M u x 0

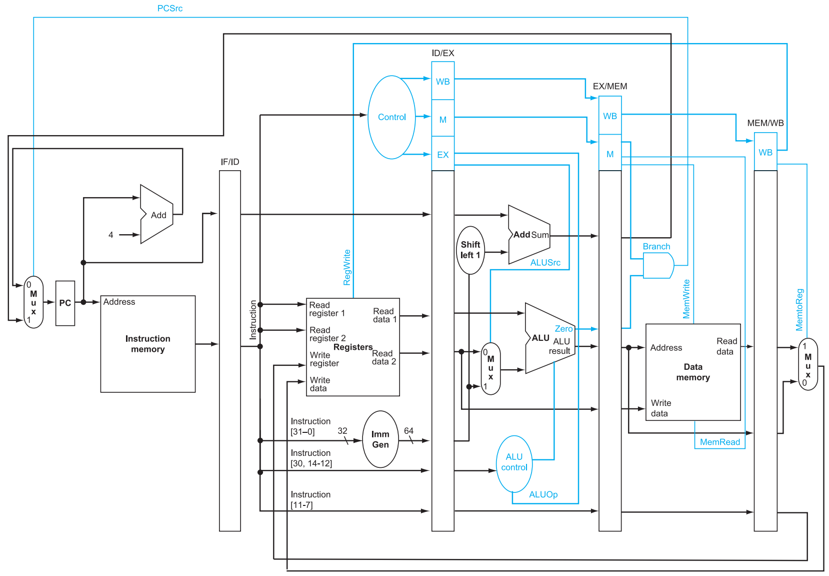
How to Wire Control Signals



How to Wire Control Signals



How to Wire Control Signals



Hazards

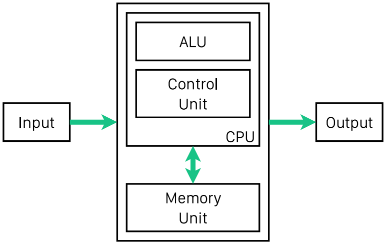
Hazards

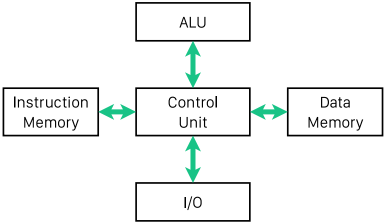
Hazards happen when the next instruction cannot execute in the next clock cycle.
There are two types of hazard

- Pipeline hazards
    - Structural hazard
    - Data hazard
- Control hazard (or Branch hazard)

Structural Hazard

Von-Neumann architecture vs. Harvard architecture





Data Hazard

When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available.

Read after Write

```
add r5, r3, r2
add r6, r5, r1
```

In this example, the second instruction READ `r5`, AFTER the first instruction WRITE it.

Data Hazard

Case 1

```
i0: add r5, r3, r2
i1: xor r6, r5, r1
i2: sub r9, r3, r5
i3: or  r2, r7, r5
i4: sll r4, r5, r5
```

IF	ID	EX	ME	WB
i2	i1	i0		
i2	i1	nop	i0	
i2	i1	nop	nop	i0
i2	i1	nop	nop	nop
i3	i2	i1	nop	nop
i4	i3	i2	i1	nop

“NOP” to stall i1 until i0 writes successfully.

Data Hazard

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB
i2	i1	i0		
i3	i2	i1	i0	
i3	i2	nop	i1	i0
i3	i2	nop	nop	i1
i3	i2	nop	nop	nop
i4	i3	i2	nop	nop

“NOP” to stall i2 until i1 writes successfully.

Control Hazard

Case 3

```
i0: add r4, r3, r2
i1: beq r5, r6, _L0
i2: sub r9, r5, r1
...
i8: _L0
    sll r4, r5, r1
i9: xor r6, r8, r2
```

IF	ID	EX	ME	WB
i2	i1	i0		
i3	i2	i1	i0	
...	i3	i2	i1	i0

If instruction `i1` is NOT TAKEN, there is nothing to worry about.

Data Hazard

Case 3

```
i0: add r4, r3, r2
i1: beq r5, r6, _L0
i2: sub r9, r5, r1
...
i8: _L0
    sll r4, r5, r1
i9: xor r6, r8, r2
```

IF	ID	EX	ME	WB
i2	i1	i0		
i3	i2	i1	i0	
i8	nop	nop	i1	i0
i9	i8	nop	nop	i1
	i9	i8	nop	nop
		i9	i8	nop

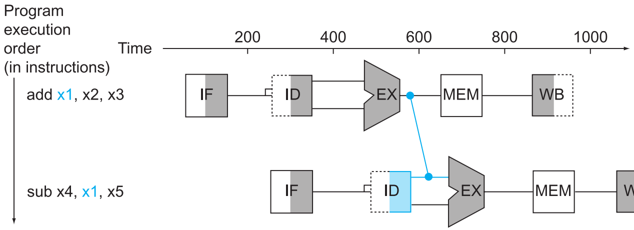
But in the other case, i2 and i3 must be flushed.

How to Stall and Flush

Enable and Reset of Flipflop.

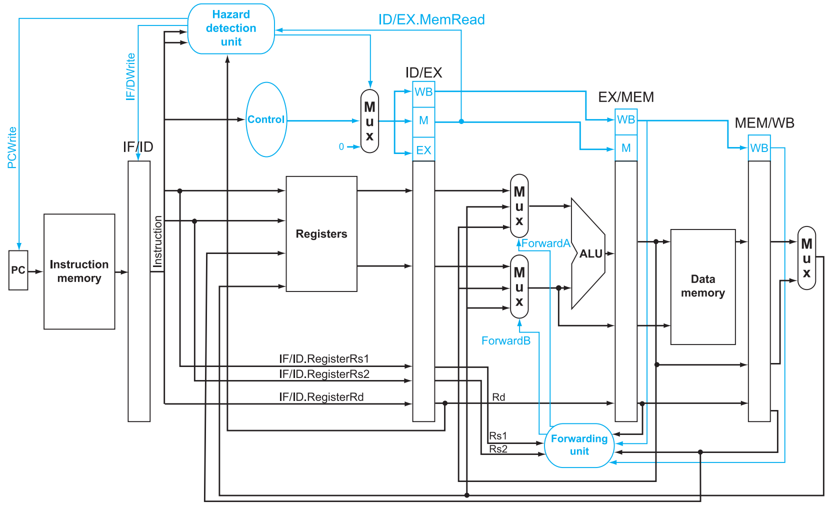
Forwarding

When the second instruction is in the pipeline, register $x1$ isn't updated, yet its value is in the pipeline already.



Forwarding — bypassing — is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

Forwarding



Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
Mux
0
PC
Instruction memory
Instruction
Registers
ForwardA
ForwardB
ALU
Data memory
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd
Rd
Rs1
Rs2
Forwarding unit

Forwarding

Case 1

```
i0: add r5, r3, r2
i1: xor r6, r5, r1
i2: sub r9, r3, r5
i3: or  r2, r7, r5
i4: sll r4, r5, r5
```

IF	ID	EX	ME	WB

Forwarding

```
// 00: no forward
// 01: forward from WB
// 10: forward from MEM
ForwardA <- 0;
if (MEM.RdWren and (MEM.RdAddr != 0) and (MEM.RdAddr ==
↪  EX.Rs1Addr))
  ForwardA <- 10;
if (WB.RdWren and (WB.RdAddr != 0) and (WB.RdAddr == EX.Rs1Addr))
  ForwardA <- 01;
```

What if both conditions are true?

Forwarding

```
// 00: no forward
// 01: forward from WB
// 10: forward from MEM
ForwardA <- 0;
if (MEM.RdWren and (MEM.RdAddr != 0) and (MEM.RdAddr ==
↪  EX.Rs1Addr))
  ForwardA <- 10;
else if (WB.RdWren and (WB.RdAddr != 0) and (WB.RdAddr ==
↪  EX.Rs1Addr))
  ForwardA <- 01;
```

Forwarding

Case 1

```
i0: add r5, r3, r2
i1: xor r6, r5, r1
i2: sub r9, r3, r5
i3: or  r2, r7, r5
i4: sll r4, r5, r5
```

IF	ID	EX	ME	WB
i2	i1	i0		
i3	i2	i1	i0	
i4	i3	i2	i1	i0
	i4	i3	i2	i1
		i4	i3	i2
			i4	i3

Forwarding resolves hazards of case 1.

Forwarding

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB

Forwarding

```
MEM.enable <- 1;
if (MEM.RdWren and (MEM.RdAddr != 0) and MEM.isload and
↪  ((MEM.RdAddr == EX.Rs1Addr) or (MEM.RdAddr == EX.Rs2Addr))
  MEM.enable <- 0;
```

Is this enough?

Forwarding

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB
i2	i1	i0		
i3	i2	i1	i0	
i4	i3	i2	i1	i0
	i4	i3	i1	i1

Where did instruction i2 go?

Forwarding

```
x.enable <- 1;
if (MEM.RdWren and (MEM.RdAddr != 0) and MEM.isload and
↪  ((MEM.RdAddr == EX.Rs1Addr) or (MEM.RdAddr == EX.Rs2Addr))
  MEM.enable <- 0;
  ID.enable  <- 0;
  EX.enable  <- 0;
```

Is this enough?

Forwarding

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB
i2	i1	i0		
i3	i2	i1	i0	
i4	i3	i2	i1	i0
i4	i3	i2	i1	i1

Why did instruction i1 stay there?

Forwarding

```
x.enable <- 1;
if (MEM.RdWren and (MEM.RdAddr != 0) and MEM.isload and
↪  ((MEM.RdAddr == EX.Rs1Addr) or (MEM.RdAddr == EX.Rs2Addr))
  MEM.enable <- 0;
  MEM.reset  <- 0;
  ID.enable  <- 0;
  EX.enable  <- 0;
```

Forwarding

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB
i2	i1	i0		
i3	i2	i1	i0	
i4	i3	i2	i1	i0
i4	i3	i2	nop	i1
	i4	i3	i2	nop
		i4	i3	i2

Forwarding can't completely resolve hazards of case 2. Why?

Branch Prediction

Branch Prediction

Branch Prediction is a technique which predicts the next instruction when the processor faces a branch instruction.
The lower the miss rate, the lower the consumed power

Let's do a little bit math

There are three types of branches, and each has its own "taken" probability.

- Unconditional branch (jumps): $P_{jump} = 100\%$
- Forward conditional branch (`if/else`): $P_{fw} = 50\%$
- Backward conditional branch (`do/while`): $P_{bw} = 90\%$

Because $P_{jump} = 100\%$, let's consider the other two only.
Given: The number of forward branches and that of backward branches are equal. The probability of a banch to be taken:

$$P = \frac{P_{fw} + P_{bw}}{2} = \frac{50\% + 90\%}{2} = 70\% \tag{1}$$

Let’s do a little bit math

IPC — instructions per cycles — indicates the performace of a processor. If a pipelined processor has too many “nop”, IPC will decrease.

Given: The number of branch instructions is 20% (P_{br}), and the processor takes TWO-CYCLE penalty or delay for a branch instruction (Δ).

$$\mathrm{IPC} = \frac{N_{instr}}{N_{cycle}} = \frac{1}{P_{br}\Delta + 1} = \frac{1}{20\% \times 2 + 1} = 71\%$$

Static Prediction

However, because we do NOT need to delay if a branch instruction is not taken, (1) shows that $P_{mis} = 70\%$, so:

$$\mathrm{IPC} = \frac{N_{instr}}{N_{cycle}} = \frac{1}{P_{br} P_{mis} \Delta + 1} = \frac{1}{20\% \times 70\% \times 2 + 1} = 78\%$$

Static Prediction

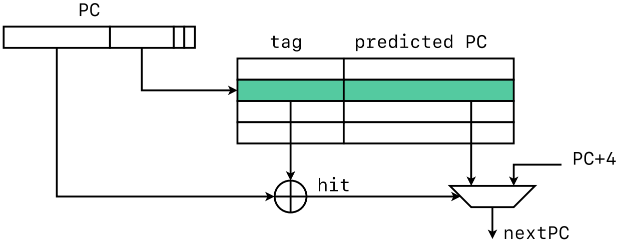
But if we know the next PC of a branch instruction and allow the processor to always go to that PC, $P_{mis} = 30\%$, so:

$$\text{IPC} = \frac{N_{instr}}{N_{cycle}} = \frac{1}{P_{br} P_{mis} \Delta + 1} = \frac{1}{20\% \times 30\% \times 2 + 1} = 89\%$$

The performace is clearly enhanced.

Branch Target Buffer

BTB, Branch Target Buffer, will save predicted PCs of branch instructions, using PC as an index or address. Because the last two bits are 00, and if all the other 30 bits are used, the buffer size will be $2^{30} = 4\mathrm{GiB}$. It is big, and not all instructions are branches; it is wasteful. Thus, to ensure the correctness of a smaller buffer, only low bits are used as the index, and the rest are called tags.



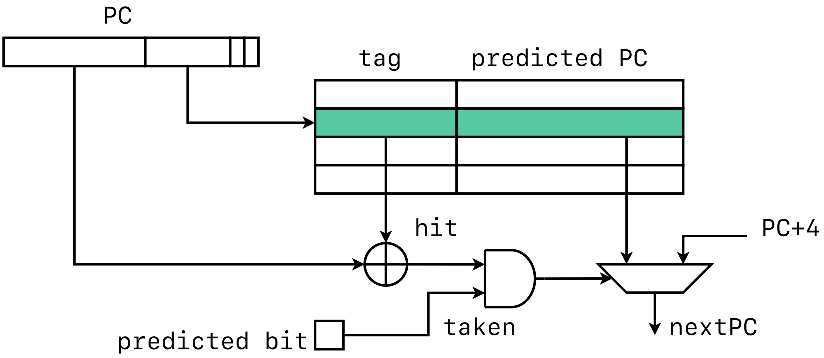
Branch Target Buffer

```
x1CA800 addi r1, r0, 30
x1CA804 addi r8, r0, 1
x1CA808 add r3, r0, r0
_COMPARE:
x1CA80C and r2, r1, r8
x1CA810 beq r2, r0, _ADD_EVEN
x1CA814 j _DECREASE
_ADD_EVEN:
x1CA818 add r3, r1, r3
_DECREASE:
x1CA81C sub r1, r1, r8
x1CA820 bne r1, r0, _COMPARE
_EXIT:
```

	tag	predicted pc
...	...	...
0x203	0x00000	0x00000000
0x204	0x001CA	0x001CA818
0x205	0x001CA	0x001CA81C
0x206	0x00000	0x00000000
0x207	0x00000	0x00000000
0x208	0x001CA	0x001CA80C
0x209	0x00000	0x00000000
...	...	...

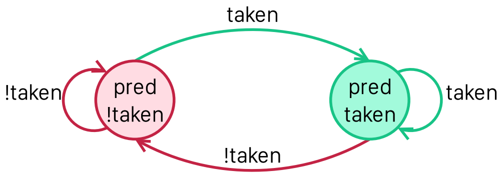
Dynamic Prediction

To increase performance, the processor needs to decide when to “taken” or not.



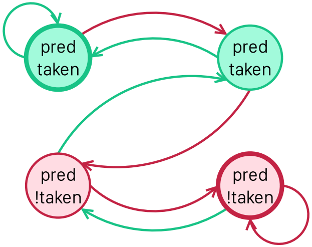
Dynamic Prediction — One Bit Prediction

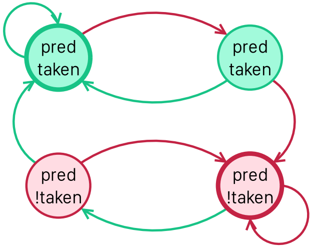
Let’s set one bit to predict taken/not taken of branch instructions, this bit simply changes its prediction when it predicts wrong. The accuracy might be 85%.



Dynamic Prediction — Two Bit Prediction

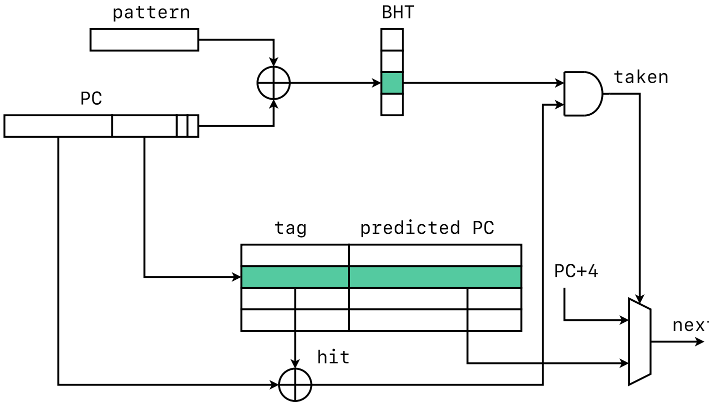
However, if the outcome is changing every time: TK–NT–TK–NT–TK–. . . , one-bit scheme becomes useless, and thus two-bit schemes become superior. The accuracy might be 90%.





Dynamic Prediction — G-share

Because the previous scheme only uses the last prediction result to predict future branches, it is not sufficent. G-share or two-bit adaptive global utilizes a pattern — a register storing branch history — to predict. In this way, the accuracy would be 93%.



Questions?