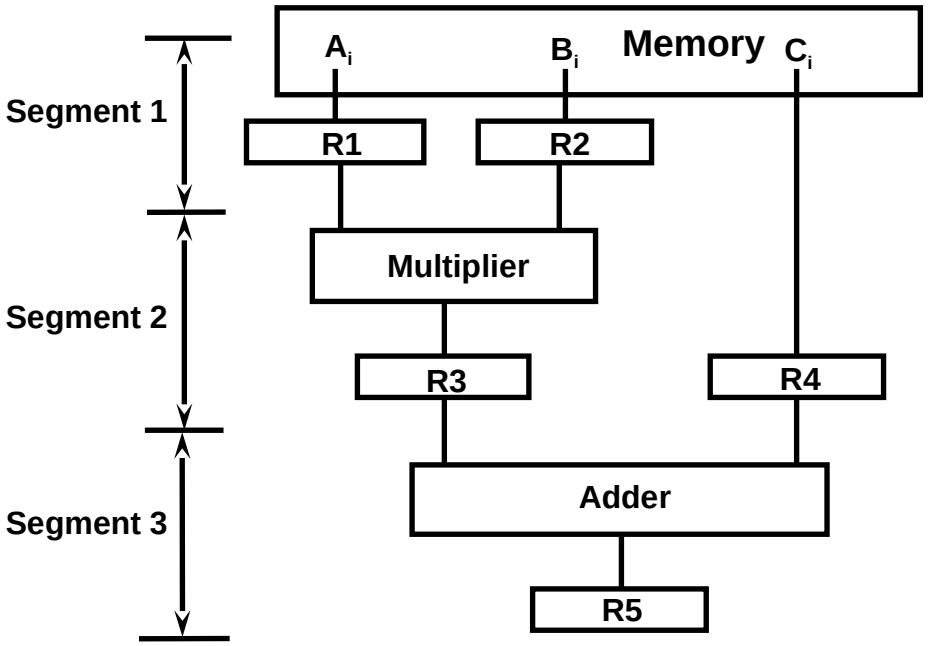
PIPELINING

A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

$A_i * B_i + C_i$ for i = 1, 2, 3, ... , 7



$R1 \leftarrow A_i$, $R2 \leftarrow B_i$ Load A_i and B_i

$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$ Multiply and load C_i

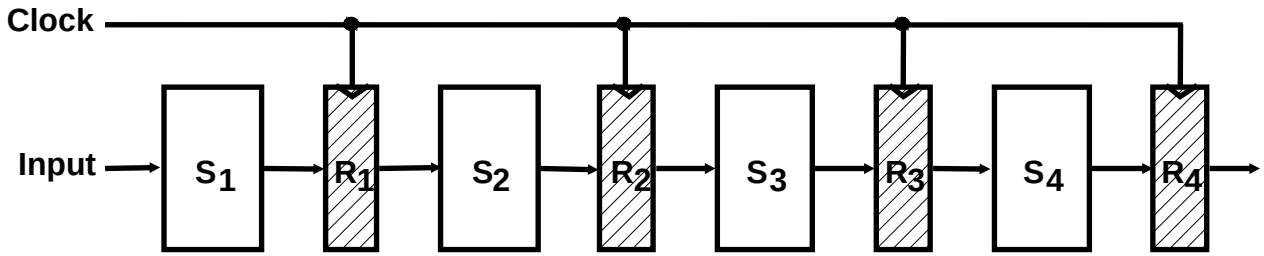
$R5 \leftarrow R3 + R4$ Add

OPERATIONS IN EACH PIPELINE STAGE

Clock Pulse Number	Segment 1		Segment 2		Segment 3	
	R1	R2	R3	R4	R5	
1	A1	B1				
2	A2		B2	A1 * B1	C1	
3	A3		B3	A2 * B2	C2	A1 * B1 + C1
4	A4		B4	A3 * B3	C3	A2 * B2 + C2
5	A5		B5	A4 * B4	C4	A3 * B3 + C3
6	A6		B6	A5 * B5	C5	A4 * B4 + C4
7	A7		B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6	
9				A7 * B7 + C7		

GENERAL PIPELINE

General Structure of a 4-Segment Pipeline



Space-Time Diagram

Segment	1	2	3	4	5	6	7	8	9
1	T1	T2	T3	T4	T5	T6			
2		T1	T2	T3	T4	T5	T6		
3			T1	T2	T3	T4	T5	T6	
4				T1	T2	T3	T4	T5	T6

Clock cycles

PIPELINE SPEEDUP

n: Number of tasks to be performed

Conventional Machine (Non-Pipelined)

t_n: Clock cycle

τ_1: Time required to complete the n tasks

$$\tau_1 = n * t_n$$

Pipelined Machine (k stages)

t_p: Clock cycle (time to complete each suboperation)

τ_k: Time required to complete the n tasks

$$\tau_k = (k + n - 1) * t_p$$

Speedup

S_k: Speedup

$$S_k = n*t_n / (k + n - 1)*t_p$$

$$\lim_{n \to \infty} S_k = \frac{t_n}{t_p} \quad (= k, \text{ if } t_n = k * t_p)$$

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

- 4-stage pipeline
- subopertion in each stage; t_p = 20nS
- 100 tasks to be executed
- 1 task in non-pipelined system; 20*4 = 80nS

Pipelined System

$(k + n - 1)*t_p = (4 + 99) * 20 = 2060nS$

Non-Pipelined System

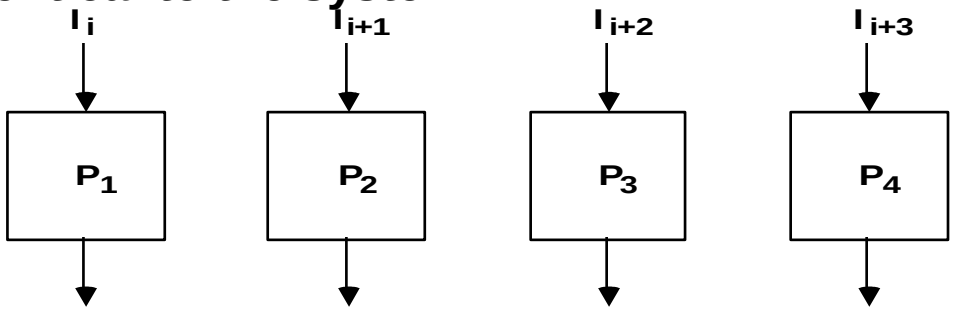
$n*k*t_p = 100 * 80 = 8000nS$

Speedup

$S_k = 8000 / 2060 = 3.88$

4-Stage Pipeline is basically identical to the system with 4 identical function units

Multiple Functional Units



INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

[1] Fetch an instruction from memory
[2] Decode the instruction
[3] Calculate the effective address of the operand
[4] Fetch the operands from memory
[5] Execute the operation
[6] Store the result in the proper place

*** Some instructions skip some phases**
*** Effective address calculation can be done in the part of the decoding phase**
*** Storage of the operation result into a register is done automatically in the execution phase**

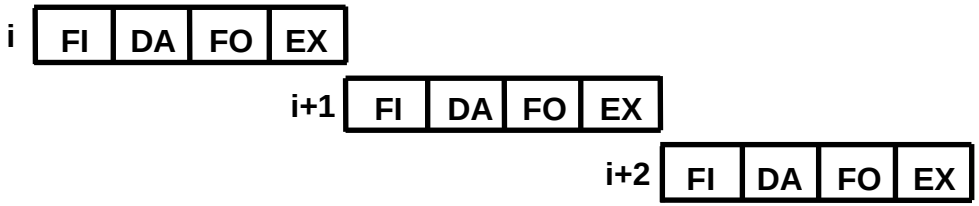
==> 4-Stage Pipeline

[1] FI: Fetch an instruction from memory
[2] DA: Decode the instruction and calculate the effective address of the operand
[3] FO: Fetch the operand
[4] EX: Execute the operation

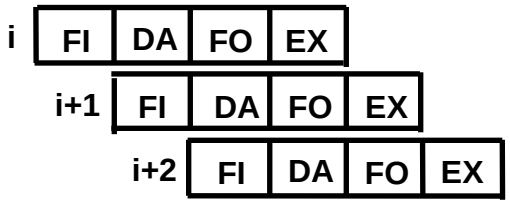
INSTRUCTION PIPELINE

Execution of Three Instructions in a 4-Stage Pipeline

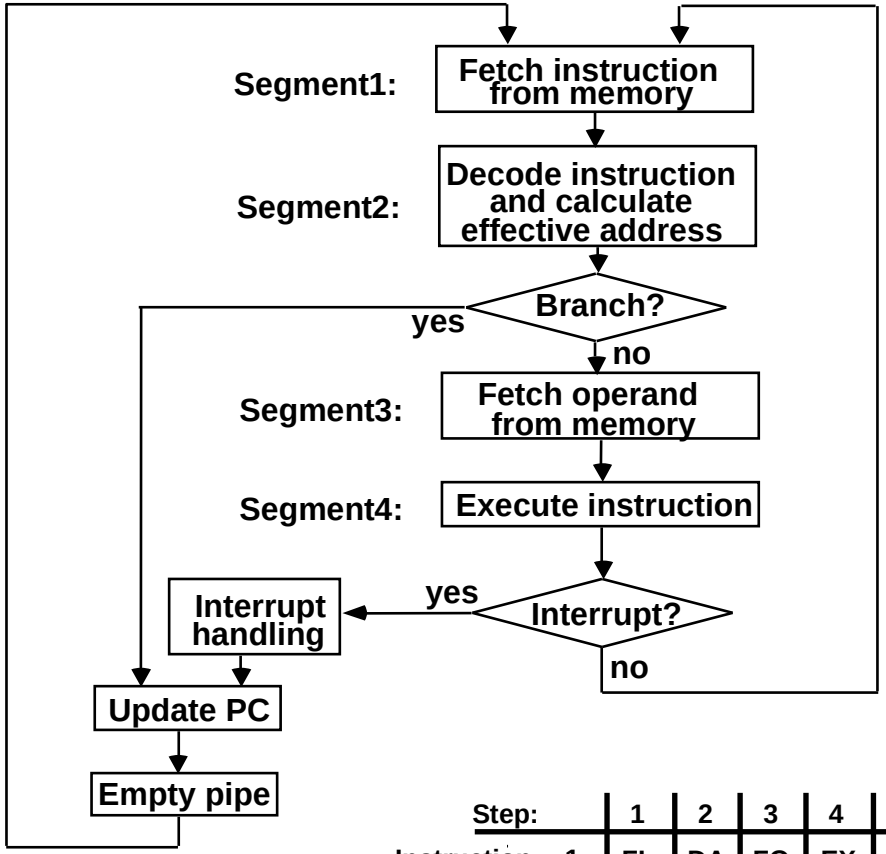
Conventional



Pipelined



INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

MAJOR HAZARDS IN PIPELINED EXECUTION

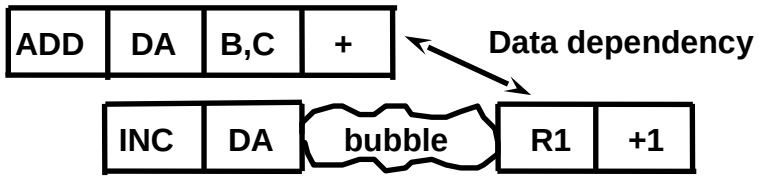
Structural hazards(Resource Conflicts)

Hardware Resources required by the instructions in simultaneous overlapped execution cannot be met

Data hazards (Data Dependency Conflicts)

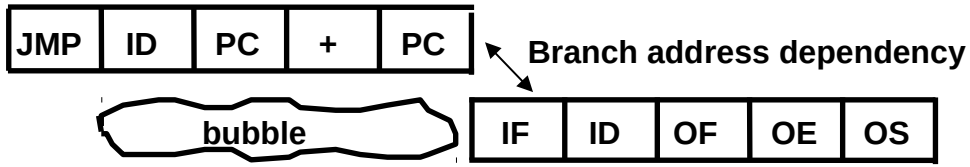
An instruction scheduled to be executed in the pipeline requires the result of a previous instruction, which is not yet available

R1 <- B + C
R1 <- R1 + 1



Control hazards

Branches and other instructions that change the PC make the fetch of the next instruction to be delayed



Hazards in pipelines may make it necessary to ***stall*** the pipeline ⇨ Pipeline Interlock: Detect Hazards Stall until it is cleared

STRUCTURAL HAZARDS

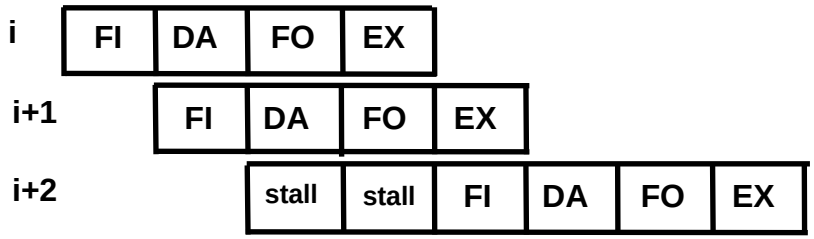
- It is caused by access to memory by two segment of pipeline at the same time.
- Some pipeline processors have shared a single-mem ory pipeline for data and instructions.
- If the EX segment needs to store the result of operati on in the data memory, while at the same time FI seg ment needs to fetch the instruction from memory.
- This can be resolved by using separate instruction a nd data memories.
- To solve this hazard, we “stall” the pipeline until the r esource is freed
- A stall is commonly called pipeline bubble, since it p asses through the pipeline taking space but carry no useful work

STRUCTURAL HAZARDS

Structural Hazards

Occur when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute

Example: With one memory-port, a data and an instruction fetch cannot be initiated in the same clock



The Pipeline is stalled for a structural hazard
<- Two Loads with one port memory
-> Two-port memory will serve without stall

Data Dependencies

- Three types of data dependencies defined in terms of how succeeding instruction depends on preceding in struction
  - RAW: Read after Write or Flow dependency
  - WAR: Write after Read or anti dependency
  - WAW: Write after Write or output dependency

Three Generic Data Hazards

- **Read After Write (RAW)**
  $Instr_J$ tries to read operand before $Instr_I$ writes it.
- **`I: add r1,r2,r3`**
- **`J: sub r4,r1,r3`**
- **R(I) corresponds to the output set**
- **D(J) corresponds to the input set**
- **$R(I) \cap D(J) \neq \varnothing$**

Write After Read (WAR)

- Write After Read (WAR)
- $Instr_J$ writes operand ***before*** $Instr_I$ reads it
  - `I: sub r4,r1,r3`
  - `J: add r1,r2,r3`
- Called as "anti-dependence"
- $D(I) \cap R(J) \neq \varnothing$

Write After Write (WAW)

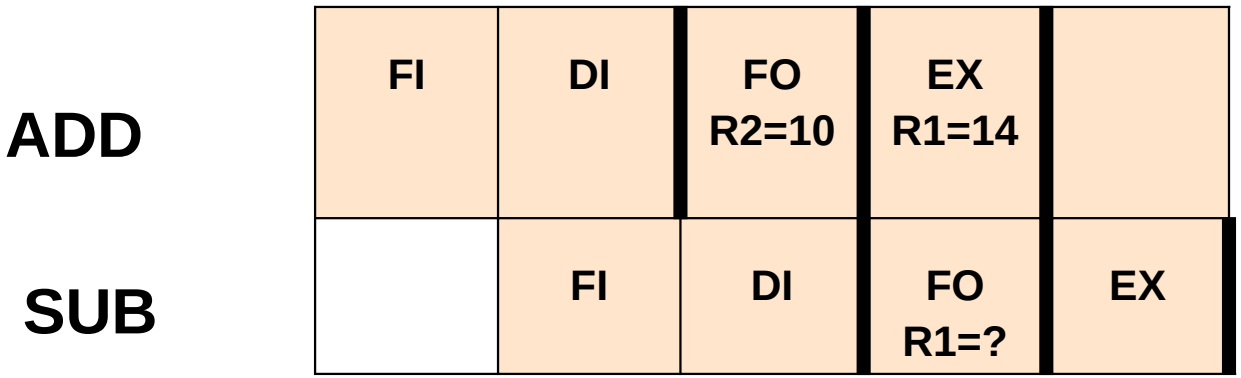
- $Instr_J$ writes operand ***before*** $Instr_I$ writes it.
  - `I: sub r1,r4,r3`
  - `J: add r1,r2,r3`
- Called as "output dependence"
- $R(I) \cap R(J) \neq \varnothing$

Data hazard

Example:

ADD R1□R2+4

SUB R4□R1-R5



Delayed load

- **Delayed load approach inserts a no-operation instruction to avoid the data conflict**

ADD R1←R2+R3
NOP
SUB R4←R1-R5

ADD	**FI**	**DI**	**FO R2=10**	**EX R1=14**		
		-	-	-	-	
NOP			**FI**	**DI**	**FO R1=14**	**EX**
SUB						

DATA HAZARDS

Data Hazards

Occurs when the execution of an instruction depends on the results of a previous instruction

```
ADD     R1, R2, R3
SUB     R4, R1, R5
```

Data hazard can be dealt with either hardware techniques or software technique

Hardware Technique

Interlock

- hardware detects the data dependencies and delays the scheduling of the dependent instruction by stalling enough clock cycles

***Forwarding* (bypassing, short-circuiting)**

- Accomplished by a data path that routes a value from a source (usually an ALU) to a user, bypassing a designated register. This allows the value to be produced to be used at an earlier stage in the pipeline than would otherwise be possible

Software Technique

Instruction Scheduling(compiler) for *delayed load*

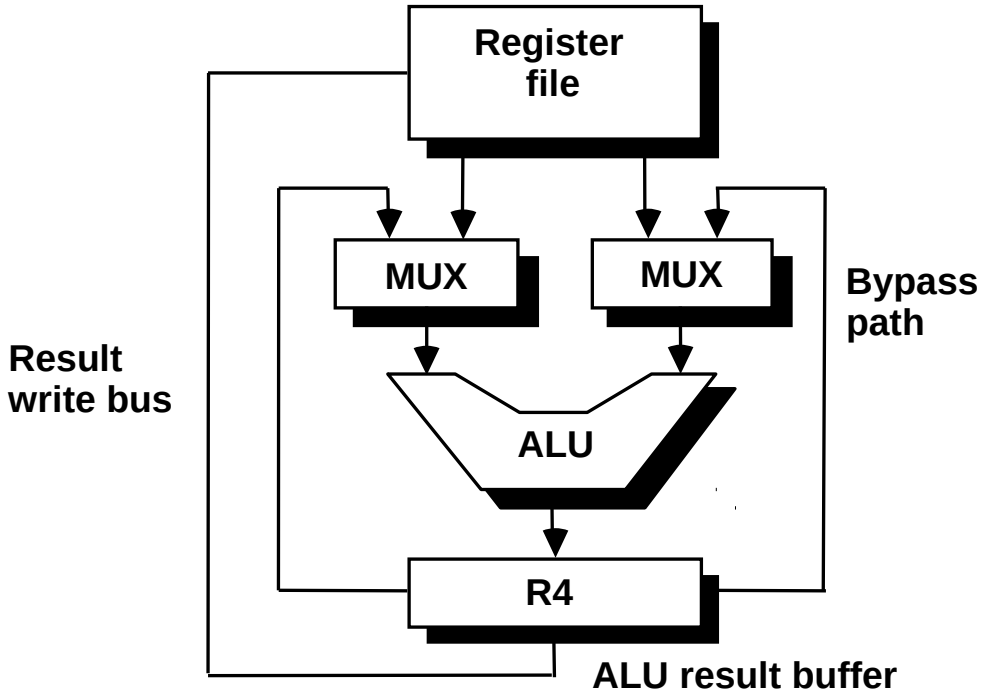
FORWARDING HARDWARE

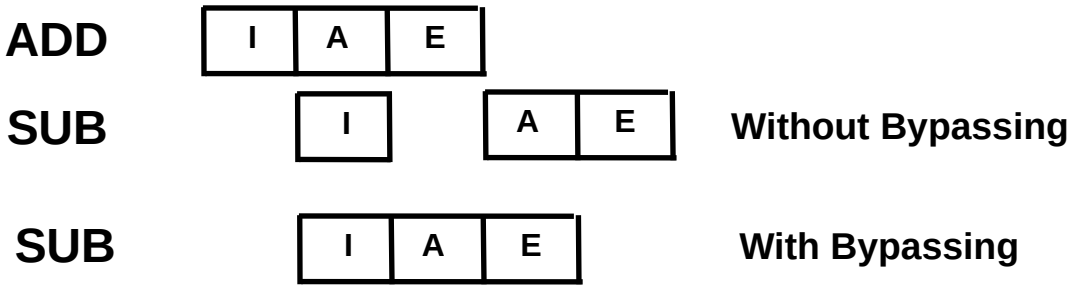
Example:

ADD R1, R2, R3
SUB R4, R1, R5

3-stage Pipeline

I: Instruction Fetch
A: Decode, Read Registers, ALU Operations
E: Write the result to the destination register





INSTRUCTION SCHEDULING

```
a = b + c;
d = e - f;
```

Unscheduled code:

```
      LW    Rb, b
      LW    Rc, c
→     ADD   Ra, Rb, Rc
→     SW    a, Ra
      LW    Re, e
      LW    Rf, f
→     SUB   Rd, Re, Rf
→     SW    d, Rd
```

Scheduled Code:

```
      LW    Rb, b
      LW    Rc, c
      LW    Re, e
      ADD   Ra, Rb, Rc
      LW    Rf, f
      SW    a, Ra
      SUB   Rd, Re, Rf
→     SW    d, Rd
```

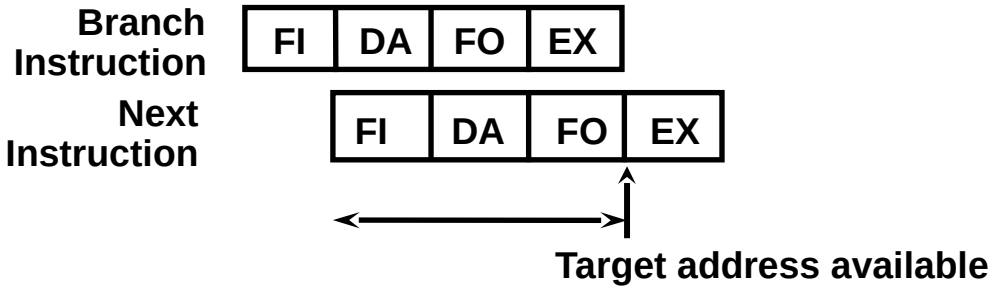
Delayed Load

A load requiring that the following instruction not use its result

CONTROL HAZARDS

Branch Instructions

- Branch target address is not known until the branch instruction is completed



- Stall -> waste of cycle times

Dealing with Control Hazards

* Prefetch Target Instruction
* Branch Target Buffer
* Loop Buffer
* Branch Prediction
* Delayed Branch

CONTROL HAZARDS

Prefetch Target Instruction

- Fetch instructions in both streams, branch not taken and branch taken
- Both are saved until branch branch is executed. Then, select the right instruction stream and discard the wrong stream

Branch Target Buffer(BTB; Associative Memory)

- Entry: Addr of previously executed branches; Target instruction and the next few instructions
- When fetching an instruction, search BTB.
- If found, fetch the instruction stream in BTB;
- If not, new stream is fetched and update BTB

Loop Buffer(High Speed Register file)

- Storage of entire loop that allows to execute a loop without accessing memory

Branch Prediction

- Guessing the branch condition, and fetch an instruction stream based on the guess. Correct guess eliminates the branch penalty

Delayed Branch

- Compiler detects the branch and rearranges the instruction sequence by inserting useful instructions that keep the pipeline busy in the presence of a branch instruction

Compiler optimization

(Rearrange the Instruction)

- In this procedure, the compiler detects the branch instruction and rearrange the instruction sequence by inserting useful instructions in the delayed slot to maintain continuous flow of pipeline.

Branch target buffer (BTB)

- BTB is an associative memory
- Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for the branch.
- When pipeline decodes a branch instruction, it searches BTB for target instruction.
  - If found, instruction will be fetched directly from BTB.

- 1000: R1<- Load[memory]
- 1001: R3<-R3+ R4
- 1002: JMP 2050
- 1003: R2<- R5+ R6

Address of Branch	Target Address
1002	2050

Prefetch target instruction

- Prefetch the target instruction in addition to the instruction following the branch.
- Fetch instruction in both path, branch taken and branch not taken.
- Both are saved until the branch is executed.
- Then select instruction from right path and discard the wrong path.

Branch Prediction

- A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed
- The idea is to assign a prediction bit P to the branch instruction when it is first executed.
- Then pipeline begins prefetching the instruction from the predicted path.
- A correct prediction eliminates the wasteful time caused by branch penalties.

Branch Prediction

- I: JNC 2000 <P>
- If P=1 -> prediction is to go 2000
- P predicts whether branch will occur or not.
- When loop iteration is controlled by I, once the loop e xecution path is entered P predicts that the same loo p path will be followed each time I will encountered.
- Misprediction eventually results when loop is exited, but it can be expected to be right most of the time.

COMPLEX INSTRUCTION SET COMPUTER

- These computers with many instructions and addressing mode s came to be known as Complex Instruction Set Computers (CIS C)
- One goal for CISC machines was to have a machine language in struction to match each high-level language statement type

VARIABLE LENGTH INSTRUCTIONS

- The large number of instructions and addressing modes led CISC machines to have variable length instruction formats
- The large number of instructions means a greater number of bits t o specify them
- In order to manage this large number of opcodes efficiently, they w ere encoded with different lengths:
  - More frequently used instructions were encoded using short opcodes.
  - Less frequently used ones were assigned longer opcodes.
- Also, multiple operand instructions could specify different address ing modes for each operand
  - For example,
    - Operand 1 could be a directly addressed register,
    - Operand 2 could be an indirectly addressed memory location,
    - Operand 3 (the destination) could be an indirectly addressed register.
- All of this led to the need to have different length instructions in dif ferent situations, depending on the opcode and operands used

VARIABLE LENGTH INSTRUCTIONS

- For example, an instruction that only specifies register operands may only be two bytes in length
  - One byte to specify the instruction and addressing mode
  - One byte to specify the source and destination registers.
- An instruction that specifies memory addresses for operands may need five bytes
  - One byte to specify the instruction and addressing mode
  - Two bytes to specify each memory address
    - Maybe more if there's a large amount of memory.
- Variable length instructions greatly complicate the fetch and decode problem for a processor
- The circuitry to recognize the various instructions and to properly fetch the required number of bytes for operands is very complex

COMPLEX INSTRUCTION SET COMPUTER

- Another characteristic of CISC computers is that they have instruc tions that act directly on memory addresses
  - For example,

    ADD L1, L2, L3

    that takes the contents of *M*[*L1*] adds it to the contents of *M*[*L2*] and stores the result in location *M*[*L3*]
- An instruction like this takes three memory access cycles to exec ute
- That makes for a potentially very long instruction execution cycle
- The problems with CISC computers are
  - The complexity of the design may slow down the processor,
  - The complexity of the design may result in costly errors in the processor desig n and implementation,
  - Many of the instructions and addressing modes are used rarely, if ever

SUMMARY: CRITICISMS ON CISC

High Performance General Purpose Instructions

- Complex Instruction
  - → Format, Length, Addressing Modes
  - → Complicated instruction cycle control due to the complex decoding HW and decoding process

- Multiple memory cycle instructions
  - → Operations on memory data
  - → Multiple memory accesses/instruction

- Microprogrammed control is necessity
  - → Microprogram control storage takes substantial portion of CPU chip area
  - → Semantic Gap is large between machine instruction and microinstruction

- General purpose instruction set includes all the features required by individually different applications
  - → When any one application is running, all the features required by the other applications are extra burden to the application

REDUCED INSTRUCTION SET COMPUTERS

- In the late ‘70s and early ‘80s there was a reaction to the shortc omings of the CISC style of processors
- Reduced Instruction Set Computers (RISC) were proposed as a n alternative
- The underlying idea behind RISC processors is to simplify the i nstruction set and reduce instruction execution time
- RISC processors often feature:
  - Few instructions
  - Few addressing modes
  - Only load and store instructions access memory
  - All other operations are done using on-processor registers
  - Fixed length instructions
  - Single cycle execution of instructions
  - The control unit is hardwired, not microprogrammed

REDUCED INSTRUCTION SET COMPUTERS

- Since all but the load and store instructions use only registers for operands, only a few addressing modes are needed
- By having all instructions the same length, reading them in is eas y and fast
- The fetch and decode stages are simple, looking much more like Mano's Basic Computer than a CISC machine
- The instruction and address formats are designed to be easy to d ecode
- Unlike the variable length CISC instructions, the opcode and regi ster fields of RISC instructions can be decoded simultaneously
- The control logic of a RISC processor is designed to be simple an d fast
- The control logic is simple because of the small number of instru ctions and the simple addressing modes
- The control logic is hardwired, rather than microprogrammed, be cause hardwired control is faster

ARCHITECTURAL METRIC

A ← B + C
B ← A + C
D ← D - B

- Register-to-register (Reuse of operands)

8	4	16
Load	rB	B
Load	rC	C
Add	rA	rB rC
Store	rA	A
Add	rB	rA rC
Store	rB	B
Load	rD	D
Sub	rD	rD rB
Store	rD	D

I = 228b
D = 192b
M = 420b

- Register-to-register (Compiler allocates operands in registers)

8	4	4	4
Add	rA	rB	rC
Add	rB	rA	rC
Sub	rD	rD	rB

I = 60b
D = 0b
M = 60b

- Memory-to-memory

8	16	16	16
Add	B	C	A
Add	A	C	B
Sub	B	D	D

I = 168b
D = 288b
M = 456b

CHARACTERISTICS OF INITIAL RISC MACHINES

	IBM 801	RISC I	MIPS
Year	1980	1982	1983
Number of instructions	120	39	55
Control memory size	0	0	0
Instruction size (bits)	32	32	32
Technology	ECL MSI	NMOS VLSI	NMOS VLSI
Execution model	reg-reg	reg-reg	reg-reg

CHARACTERISTICS OF RISC

- **RISC Characteristics**
  - **Relatively few instructions**
  - **Relatively few addressing modes**
  - **Memory access limited to load and store instructions**
  - **All operations done within the registers of the CPU**
  - **Fixed-length, easily decoded instruction format**
  - **Single-cycle instruction format**
  - **Hardwired rather than microprogrammed control**

- **Advantages of RISC**
  - **VLSI Realization**
  - **Computing Speed**
  - **Design Costs and Reliability**
  - **High Level Language Support**

ADVANTAGES OF RISC

- **VLSI Realization**

  Control area is considerably reduced

  Example:
  RISC I: 6%
  RISC II: 10%
  MC68020: 68%
  general CISCs: ~50%

  ⇒ RISC chips allow a large number of registers on the chip

  - Enhancement of performance and HLL support
  - Higher regularization factor and lower VLSI design cost

  The GaAs VLSI chip realization is possible

- **Computing Speed**

  - Simpler, smaller control unit ⇒ faster
  - Simpler instruction set; addressing modes; instruction format
  ⇒ faster decoding
  - Register operation ⇒ faster than memory operation
  - Register window ⇒ enhances the overall speed of execution
  - Identical instruction length, One cycle instruction execution
  ⇒ suitable for pipelining ⇒ faster

ADVANTAGES OF RISC

- **Design Costs and Reliability**

  - Shorter time to design
    ⇒ reduction in the overall design cost and
    reduces the problem that the end product will
    be obsolete by the time the design is completed

  - Simpler, smaller control unit
    ⇒ higher reliability

  - Simple instruction format (of fixed length)
    ⇒ ease of virtual memory management

- **High Level Language Support**

  - A single choice of instruction
    ⇒ shorter, simpler compiler

  - A large number of CPU registers
    ⇒ more efficient code

  - Register window
    ⇒ Direct support of HLL

  - Reduced burden on compiler writer