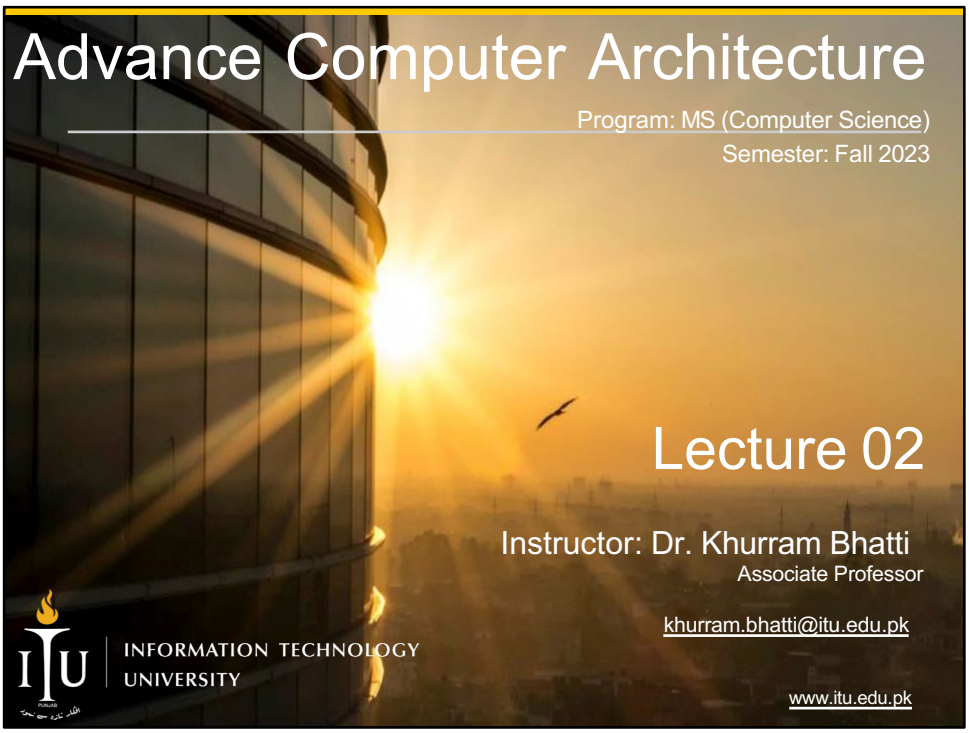


Advanced Computer Architecture

- Instruction Set Architecture (ISA)
  - The Seven Dimensions
- RISC Architecture
  - RISC vs CISC architecture
- RISC Assembly Language
- Basic Concepts of Pipelining

Advanced Computer Architecture

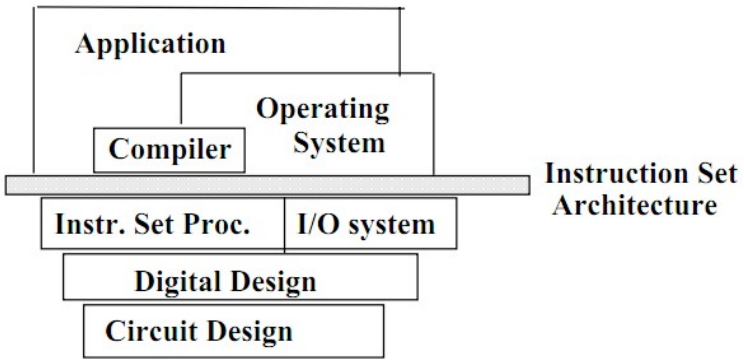
Instruction Set Architecture (ISA)

- By Definition
  - Instruction Set Architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine
- What ISA defines
  - Operations that the processor can execute
  - Data Transfer + Access mechanisms for data
  - Control Mechanisms (branch, jump, etc)
  - Contract between programmer/compiler + HW

Advanced Computer Architecture

Instruction Set Architecture (ISA)

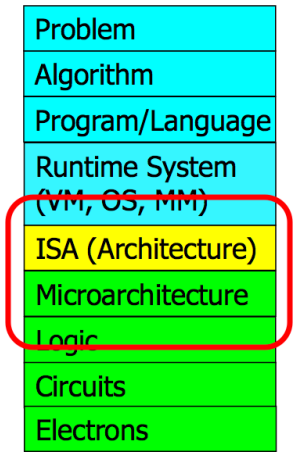
- By Definition



Advanced Computer Architecture

Instruction Set Architecture (ISA)

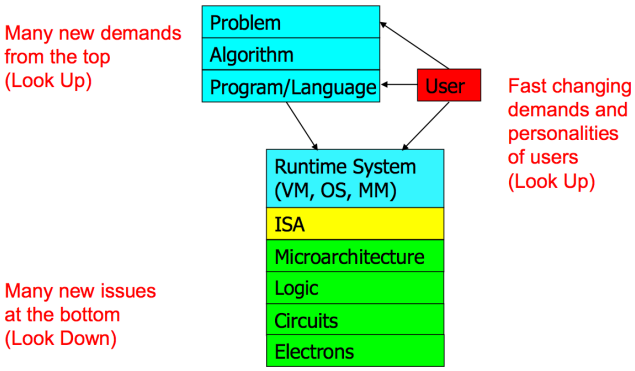
- By Definition
  - Full computing stack



Advanced Computer Architecture

Instruction Set Architecture (ISA)

- By Definition
  - Entire computing stack should be optimized!



Advanced Computer Architecture

Instruction Set Architecture (ISA)

- A good ISA
  - Portable, Compatible
    - Lasts through many implementations
  - Used in many different ways
    - PMDs, Desktops, Servers, Tablets
  - Provides convenient functionality to higher layers
  - Permits efficient implementation at lower layers

Advanced Computer Architecture

Instruction Set Architecture (ISA)

- Some Example ISAs
  - x86: Intel 8086, extended into x86-64
  - ARM: 32-bit and 64-bit architecture
  - MIPS: 32-bit and 64-bit architecture
  - SPARC: 32-bit and 64-bit architecture
  - PIC: 8-bit to 32-bit architecture
  - Z80: 8-bit architecture

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

① Which class an ISA belongs:

- Defined based on how memory is accessed
  - Almost all ISAs are general-purpose register architectures
  - Operands are registers or memory locations
- Two main types
  - Register-Memory ISAs
    - Instructions can access memory as part of their execution Example: 80x86 ISA
  - Load-Store ISAs
    - Only LOAD or STORE instructions can access memory
    - Example: ARM, MIPS
- All recent ISAs are load-store class

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

② Memory Addressing:

- Virtually all computers use byte addressing to access memory operands
- Some architectures, like ARM and MIPS, require that the data must be aligned
  - Data alignment means putting the data at a memory address equal to some *multiple* of the *word size*
- A computer reads/ writes to a memory address in *word sized* chunks
  - Example: With word size of 4 bytes, the data to be read should be at a memory address, which is some multiple of 4

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

② Memory Addressing:

- Data alignment increases the system's performance due to the way the CPU handles memory
- An access to an object/data of size S bytes at byte address A is aligned if ?
- If NOT aligned: Computer has to read two or more word size chunks and do some calculation before the requested data has been read, or it may generate an alignment fault
- Solution: Data Structure Padding!
  - Some meaningless bytes between the end of the last data structure and the start of the next are inserted

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

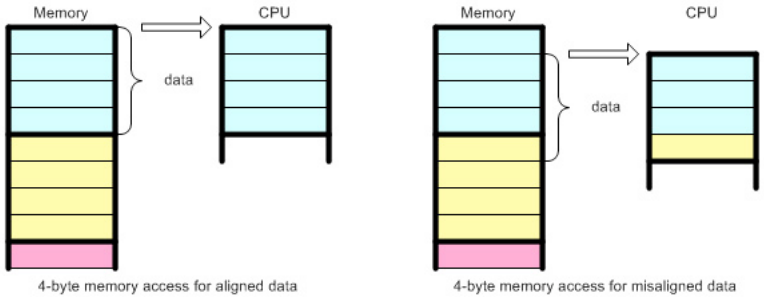
② Memory Addressing: Data Alignment

- Data (variable) has 2 properties: Value and the storage location (address)
- CPU does not read from or write to memory 1-byte at a time.
- CPU accesses memory in 2, 4, 8, 16, or 32 byte chunks at a time.
- Accessing an address on 4-byte or 16-byte boundary is a lot faster than accessing an address on 1-byte boundary
- Problem:
  - If data size is variable, the address does not remain byte-addressable
  - If the data is misaligned of 4-byte boundary, CPU has to load 2 chucks of data, shift out unwanted bytes then combine them together.

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

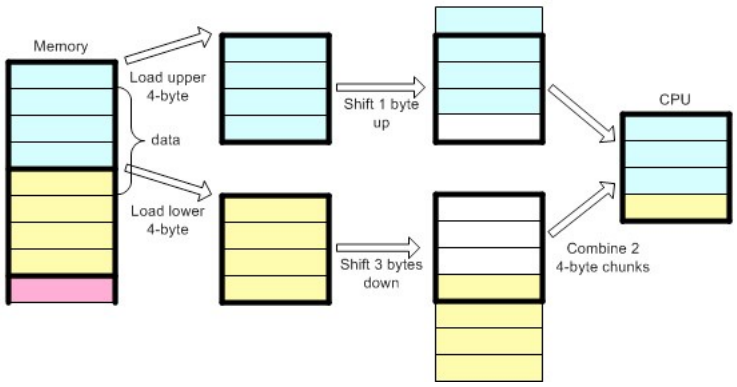
② Memory Addressing: Data Alignment



Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

② Memory Addressing: Data Alignment



Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

③ Addressing Modes:

- Define how machine language instructions in the architecture identify the operand(s) of each instruction
- Specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction
  - Various addressing modes are supported by ISAs

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

③ Addressing Modes:

Addressing mode	Example instruction	Meaning	When used
Register	Add R4, R3	Regs[R4] ← Regs[R4] + Regs[R3]	When a value is in a register.
Immediate	Add R4, #3	Regs[R4] ← Regs[R4] + 3	For constants.
Displacement	Add R4, 100(R1)	Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4, (R1)	Regs[R4] ← Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer or a computed address.
Indexed	Add R3, (R1 + R2)	Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1, (1001)	Regs[R1] ← Regs[R1] + Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1, @(R3)	Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer *p*, then mode yields **p*.
Autoincrement	Add R1, (R2)+	Regs[R1] ← Regs[R1] + Mem[Regs[R2]] Regs[R2] ← Regs[R2] + *d*	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	Add R1, –(R2)	Regs[R2] ← Regs[R2] – *d* Regs[R1] ← Regs[R1] + Mem[Regs[R2]]	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1, 100(R2)[R3]	Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * *d*]	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

④ Types & Sizes of Operands:

- Most ISAs like 80x86, ARM, and MIPS support following operand types and sizes
  - 8-bit (ASCII character)
  - 16-bit (unicode character)
  - 32-bit (integer or word)
  - 64-bit (double word or long integer)
  - Floating point 32-bit (single precision)
  - 64-bit (double precision)
  - 80-bit floating point (extended double precision) supported by 80x86 only

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

⑤ Operations:

- Almost all ISAs support general category of operations such as:
  - Data transfer operations
  - Arithmetic operations
  - Logical operations
  - Control operations
  - Floating point operations

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

⑥ Control-flow Instructions:

- All ISAs support control-flow instructions such as
  - Conditional branches
  - Unconditional jumps
  - Procedure calls and returns

Advanced Computer Architecture

Instruction Set Architecture –Seven dimensions

⑦ Encoding an ISA:

- Refers to the way a particular ISA formats its instructions
- Two basic types
  - Fixed Length: ARM and MIPS instructions are fixed length , 32-bit long, simple to encode/decode
  - Variable Length: 80x86 encoding is variable length, ranging from 1 to 18 bytes.
  - Variable- length instructions can take less space than fixed-length instructions, not easy to decode.

Advanced Computer Architecture

- Instruction Set Architecture (ISA)
  - The Seven Dimensions
- RISC Architecture
  - RISC vs CISC architecture
- RISC Assembly Language
- Basic Concepts of Pipelining

Advanced Computer Architecture

RISC Architecture

- Define RISC: Reduced Instruction Set Computer
  - A type of microprocessor architecture that utilizes a small but highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures
  - First RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s
  - Prof. John Hennessy, Stanford Univ., was the first to propose an early stage RISC

Advanced Computer Architecture

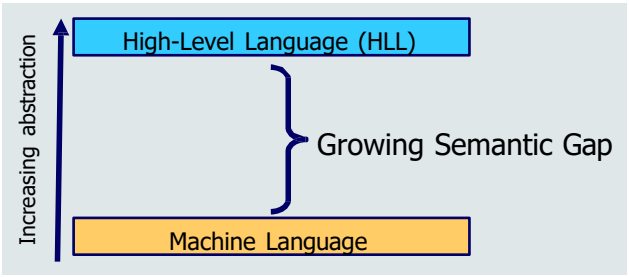
RISC Architecture

- Characteristic Design Features of RISC?
  - One cycle execution time:
    - RISC processors have a CPI (clock per instruction) of one cycle.
    - This is due to the optimization of each instruction on the CPU and pipelining
  - Large number of registers: RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory
  - Pipelining: A technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions

Advanced Computer Architecture

RISC vs CISC Architecture

- Both RISC and CISC architectures have been developed to reduce the semantic gap
- Semantic gap:



Advanced Computer Architecture

RISC vs CISC Architecture

- Semantic gap:
  - Efficiency of software development is improved through powerful high-level programming languages (e.g., Ada, C++, Java)
  - HLLs support higher levels of abstraction
  - Evolution in HLLs has increased the semantic gap between programming languages and machine languages

Advanced Computer Architecture

RISC vs CISC Architecture

- Main Features of CISC
  - CISC attempts to make machine language instructions similar to HLL statements
  - A large number of instructions (> 200) with complex instructions and data types
  - Many and complex addressing modes (e.g., indirect addressing is often used)
  - Memory bottleneck is a major problem
    - Due to complex addressing modes and multiple memory accesses instruction

Advanced Computer Architecture

RISC vs CISC Architecture

- Not constrained to load/store architecture
- Instructions may be larger than one word
- Typically use two-operand instruction format, with at least one operand in a register
- Implementation of C = A + B using CISC:

```
Move    Ri, A
Add     Ri, B
Move    C, Ri
```


Advanced Computer Architecture

RISC vs CISC Architecture

- Problems with CISC
  - Large instruction set requires complex and time consuming hardware steps to decode and execute instructions
  - Complex machine instructions may not match HLL statements exactly, in which case they may be of little use
  - Instruction sets designed with specialized instructions for several HL languages will not be efficient when executing program of a given language
  - It will lead also to a complex design tasks, thus large time-to-market.

Advanced Computer Architecture

RISC vs CISC Architecture

- Main Features of RISC
  - Small number of simple instructions (desirably < 100)
    - Simple and small decode and execution hardware is required
    - CPU takes less silicon area to implement, and runs also faster
  - Execution of one instruction per clock cycle
  - The instruction pipeline performs more efficiently due to simple instructions and similar execution patterns
  - Complex operations are executed as a sequence of simple instructions
    - In the case of CISC they are executed as one single or a few complex instructions

Advanced Computer Architecture

RISC vs CISC Architecture

- Main Features of RISC (Cont'd)
  - LOAD-and-STORE architecture
    - Only LOAD and STORE instructions reference data in memory
    - All other instructions operate only with registers (i.e., register-to- register instructions)
  - Only a few simple addressing modes are used.
    - Example: register, direct, register indirect…
  - Instructions are of fixed length and uniform format
    - Loading and decoding of instructions are simple and fast.
    - It is not needed to wait until the length of an instruction is known in order to start decoding it
    - Decoding is simplified because the opcode and address fields are located in the same position for all instructions.

Advanced Computer Architecture

RISC vs CISC Architecture

- Main Features of RISC (Cont'd)
  - A large number of registers is available
    - Variables and intermediate results can be stored in registers and do not require repeated loads and stores from/to memory
    - All local variables of procedures and the passed parameters can be stored in registers

Advanced Computer Architecture

RISC vs CISC Architecture

- Consider the same high-level language statement:

  C = A + B

- A, B, and C correspond to memory locations
- STEPS:
  - Fetch contents of locations A and B
  - Compute sum
  - Transfer result to location C

Advanced Computer Architecture

RISC vs CISC Architecture

- Sequence of simple RISC instructions for task:

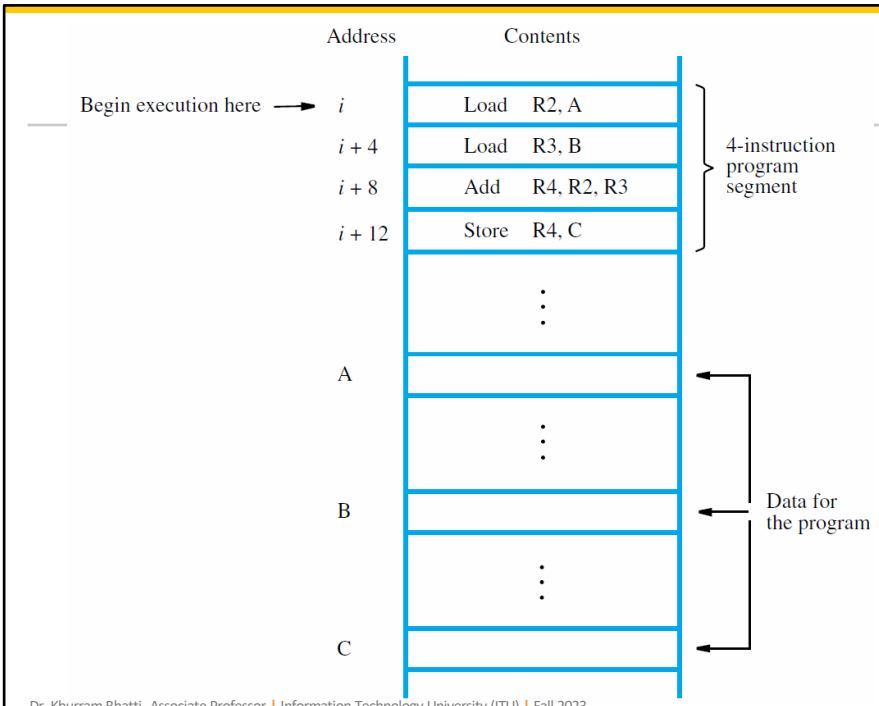
```
Load    R2, A
Load    R3, B
Add     R4, R2, R3
Store   R4, C
```

- Load instruction transfers data to register
- Store instruction transfers data to the memory
- Destination differs with same operand order

Advanced Computer Architecture

RISC vs CISC Architecture

- Consider the preceding 4-instruction program
- How is it stored in the memory?
  (32-bit word length, byte-addressable)
- Place first RISC instruction word at address *i*
- Remaining instructions are at:
  i + 4
  i + 8
  i + 12



Advanced Computer Architecture

RISC vs CISC Architecture

- Problems with RISC
  - An operation might need two, three, or more instructions to accomplish
    - More memory access might be needed.
    - Execution speed may be reduced for certain applications
  - It usually leads to longer programs, which needs larger memory space to store.

Advanced Computer Architecture

RISC vs CISC Architecture

- Summary:
  - Studies have shown that benchmark programs run often faster on RISC processors than on CISC machines
    - However, it is difficult to identify which RISC feature really produces the higher performance
  - Some "CISC fans" argue that the higher speed is not produced by the typical RISC features but because of technology, better compilers, etc.
  - The simpler RISC instruction set results in a larger memory requirement compared to the CISC case.

Advanced Computer Architecture

RISC vs CISC Architecture

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes	large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers
High cycles per second	Low cycles per second
Variable length Instructions	Equal length instructions which make pipelining possible

Advanced Computer Architecture

RISC vs CISC Architecture

- Example:

The CISC Approach

- **Instruction :**

  MULT 2:3, 5:2

Operations:

1. Loads the two operands into separate registers
2. Multiplies the operands in the execution unit
3. Then stores the product in the some temporary register
4. Stores value back to memory location 2:3

The RISC Approach

- **Instructions :**

```
LW      A, 2:3
LW      B, 5:2
MULT    A, B
SW      2:3, A
```

Operations:

1. Load operand1 into register A
2. Load operand2 into register B
3. Multiply the operands in the execution unit and store result in A
4. Store value of A back to memory location 2:3

Advanced Computer Architecture

- Instruction Set Architecture (ISA)
  - The Seven Dimensions
- RISC Architecture
  - RISC vs CISC architecture
- RISC Assembly Language
- Basic Concepts of Pipelining

ISA & Assembly Language
RISC-V Example

Programming Computers

How can we program a computer?

- Programming languages: Easily express algorithms in a human readable way. Compiler does the dirty work.
  $\Longrightarrow$ Focus on productivity
  $\Longrightarrow$ Focus on maintainability
  $\Longrightarrow$ Focus on abstractions
- Assembly language: Efficient to process by a computer, not quite readable to (most) humans.
  $\Longrightarrow$ Focus on memory requirements (code size)
  $\Longrightarrow$ Focus on hardware complexity
  $\Longrightarrow$ Provide access to the processor's capabilities

Instruction Set Architecture

Defines the *language* of a processor:

- Processor reads one instruction after another
- Each instruction is *executed*
  - May depend on instruction operands (registers, constants)
  - Correspond to simple operations (e.g., arithmetic)
  - Update the processor's internal state
  - Processor does what the instruction says
  - Processor does not *know* what he is doing
- Each instruction has a binary representation
  - For humans: each instruction has a textual form

Example: C – Assembly – Binary

- High-level C Code:

```
int foo(int a, int b, int c, int d) {
  return (a + b) - (c + d)
}
```

- RISC-V assembly code (one line per instruction):

```
add     a0,a0,a1        // a0 = (a + b)
add     a2,a2,a3        // a2 = (c + d)
sub     a0,a0,a2        // a0 = (a0 - a2)
ret                     // return
```

- RISC-V binary code (32 bits per instruction):

funct7	rs2	rs1	funct3	rd	opcode
0000000_2	01011_2	01010_2	000_2	01010_2	0110011_2
0000000_2	01101_2	01100_2	000_2	01100_2	0110011_2
0100000_2	01100_2	01010_2	000_2	01010_2	0110011_2
000000000000_2		00001_2	000_2	00000_2	1100111_2

RISC-V Processor State

View from assembly programmer:

- A single global memory
  - Each memory element has 1 byte
  - Brackets denote memory addresses ([100])
  - Stores instructions and data
  - You can see it like a simple byte array in C
- 32 general-purpose registers
  - Each register consists of 4 bytes (32 bits)
  - Register names: `x0, ..., x31`
    (conventions and pseudonyms on next slide)
- A 32-bit program counter (PC)
  - Indicating the next instruction to read
- Registers and memory may represent arbitrary data

RISC-V General-Purpose Register Names

Description	Number	Pseudonym
Zero (always gives 0)	x0	–
Return address	x1	ra
Stack pointer	x2	sp
Global pointer	x3	gp
Thread pointer	x4	tp
Temporary registers	x5 – x7	t0 – t2
Frame pointer (Saved Register)	x8	fp (or s0)
Saved register	x9	s1
Function arguments/Return values	x10 – x11	a0 – a1
Function arguments	x12 – x17	a2 – a7
Saved registers	x18 – x27	s2 – s11
Temporary registers	x28 – x31	t3 – t6

RISC-V Instruction Set

View from assembly programmer:

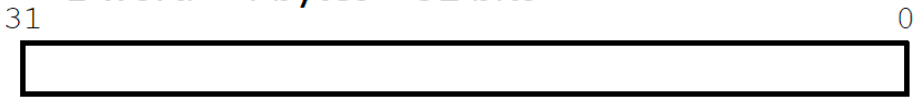
- Programs are a sequence of instructions stored in memory
- Each instruction has an address
- Each instruction has a mnemonic (opcode)
- Each instruction has operands
  - Up to three register operands: `rs1`,`rs2`,`rd` (`rs1`,`rs2` for reading, `rd` for writing)
  - An optional constant *immediate* value: `imm` (limited in number of bits)

Example: RISC-V Instructions

- Signed integer addition:
  - General format: **`add`** `rd, rs1, rs2`
  - Meaning: add value of registers `rs1` and `rs2` store result in register `rd`
  - Example: **`add`** `a0, a0, a1`
- Signed immediate integer addition:
  - General format: **`addi`** `rd, rs1, imm`
  - Meaning: add value of register `rs1` and `imm` store result in register `rd`
  - Example: **`addi`** `t0, a0, 5`

Example: RISC-V Instructions

- By convention, RISCV instructions are each 1 word = 4 bytes = 32 bits



- Divide the 32 bits of an instruction into "fields"
  - regular field sizes → simpler hardware
  - will need some variation….
- Define 6 types of *instruction formats:*
  - R-Format I-Format S-Format U-Format SB-Format UJ-Format

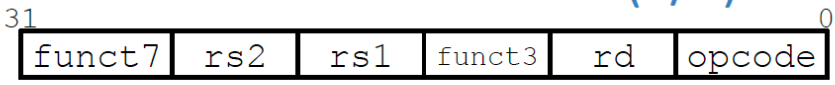
Example: RISC-V Instructions

The 6 Instruction Formats

- R-Format: instructions using 3 register inputs
  - `add`, `xor`, `mul` —arithmetic/logical ops
- I-Format: instructions with immediates, loads
  - `addi`, `lw`, `jalr`, `slli`
- S-Format: store instructions: `sw`, `sb`
- SB-Format: branch instructions: `beq`, `bge`
- U-Format: instructions with upper immediates
  - `lui`, `auipc` —upper immediate is 20-bits
- UJ-Format: jump instructions: `jal`

Example: RISC-V Instructions

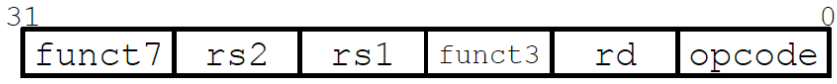
R-Format Instructions (2/3)



- `opcode` (7): partially specifies operation
  - e.g. **R**-types have opcode = 0b0110011, **SB** (branch) types have opcode = 0b1100011
- `funct7+funct3` (10): combined with `opcode`, these two fields describe what operation to perform

Example: RISC-V Instructions

R-Format Instructions (3/3)



- `rs1` (5): 1st operand ("source register 1")
- `rs2` (5): 2nd operand (second source register)
- `rd` (5): "destination register" — receives the result of computation
- **Recall:** RISCV has 32 registers
  - A 5 bit field can represent exactly 2^5 = 32 things (interpret as the register numbers **x0-x31**)

RISC-V Instruction Classes and Formats

- **R-Format**:
  - Instructions operating on registers (thus the *R*)
  - Examples: **`add`**, **`or`**, **`sll`**, **`slt`**, **`div`**, . . .

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- **I-Format**:
  - Instructions taking an immediate constant (thus the *I*)
  - Examples: **`addi`**, **`andi`**, **`srai`**, **`sltiu`**, **`lw`**, **`jalr`**, . . .

imm	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

RISC-V Instruction Classes and Formats

- **S** and **SB-Format**:
  - Stores/cond. branches with an immediate constant
  - Examples: **`sw`**, **`beq`**, . . .

imm	rs2	rs1	funct3	imm	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- **U** and **UJ-Format**:
  - Instructions/jumps taking a large immediate constant
  - Examples: **`jal`**, **`lui`**, . . .

imm	rd	opcode
20 bits	5 bits	7 bits

Example: RISC-V R-Format

- **`add`** `a0, a0, a1` (recall: `a0=x10`, `a1=x11`)

0	11	10	0	10	51
funct7	rs2	rs1	funct3	rd	opcode

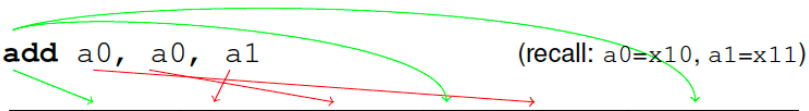
Example: RISC-V R-Format

- **add** a0, a0, a1 (recall: a0=x10, a1=x11)

0000000_2	01011_2	01010_2	000_2	01010_2	0110011_2
funct7	rs2	rs1	funct3	rd	opcode

Example: RISC-V R-Format

- **add** a0, a0, a1 (recall: a0=x10, a1=x11)



0000000_2	01011_2	01010_2	000_2	01010_2	0110011_2
funct7	rs2	rs1	funct3	rd	opcode

Example: RISC-V I-Format

- **addi** `t0, a0, 5` (recall: `t0=x5, a0=x10`)

5	10	0	5	19
imm	rs1	funct3	rd	opcode

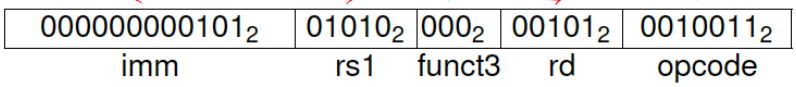
Example: RISC-V I-Format

- **addi** `t0, a0, 5` (recall: `t0=x5, a0=x10`)

000000000101_2	01010_2	000_2	00101_2	0010011_2
imm	rs1	funct3	rd	opcode

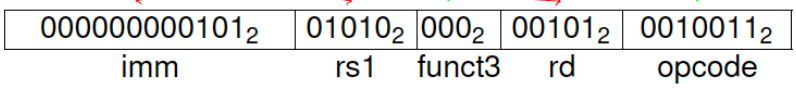
Example: RISC-V I-Format

- **addi** t0, a0, 5 (recall: t0=x5, a0=x10)

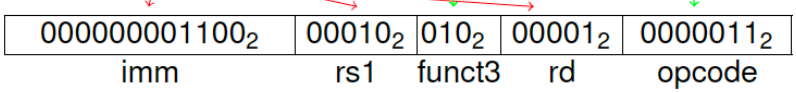


Example: RISC-V I-Format

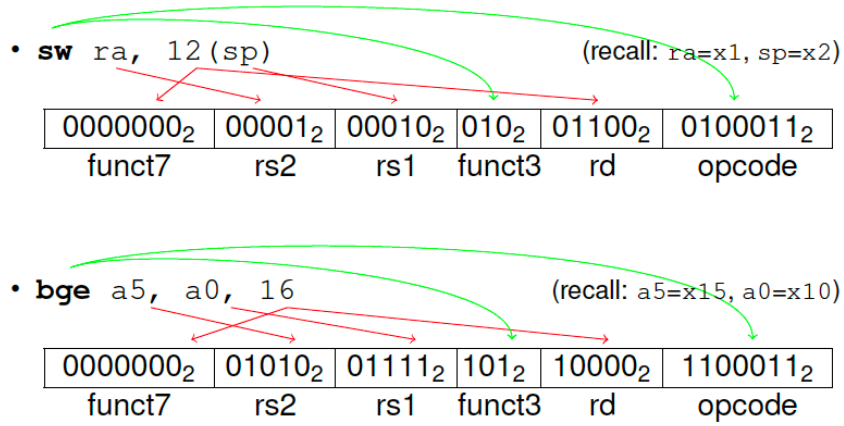
- **addi** t0, a0, 5 (recall: t0=x5, a0=x10)



- **lw** ra, 12(sp) (recall: ra=x1, sp=x2)



Example: RISC-V S- and SB-Formats



Example: RISC-V S- and SB-Formats

Branch Instructions [beq, bne, bge, …]

– Need to specify an address to go to
– Also take two registers to compare
– Doesn't write into a register (No destination register!)

If we don't take the branch:

```
PC = PC+4 = next instruction
```

If we do take the branch:

```
PC = PC + (immediate*4)
```

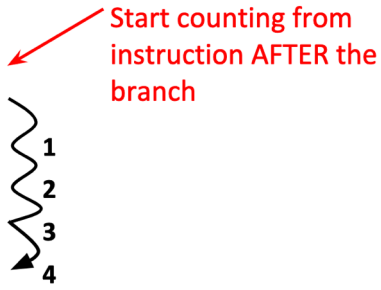
Observations: `immediate` is number of instructions to move either forward (+) or backwards (–)

Example: RISC-V S- and SB-Formats

Example:

- RISCV Code:

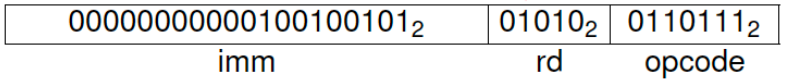
```
Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  <target instr>
```



- Branch offset = 4×32-bit instructions = 16 bytes
- (Branch with offset of 0, branches to itself)

Example: RISC-V U- and UJ-Formats

- **lui** a0, 293 (recall: ra=x1, sp=x2)



Pipelining

Computer Architecture & Assembly

Basic Pipelining

- By Definition:
  - An implementation technique whereby multiple instructions are overlapped in execution
  - It takes advantage of parallelism that exists among the actions needed to execute an instruction
- Fundamental Advantage:
  - Increased throughput –the number of instructions completed per unit of time
  - DO NOT CONFUSE: It does not reduce the execution time of an individual instruction!

Computer Architecture & Assembly

Basic Pipelining

- By Example: Laundry pipelining

04 loads for laundry

Washer takes 30-min

Dryer takes 30-min

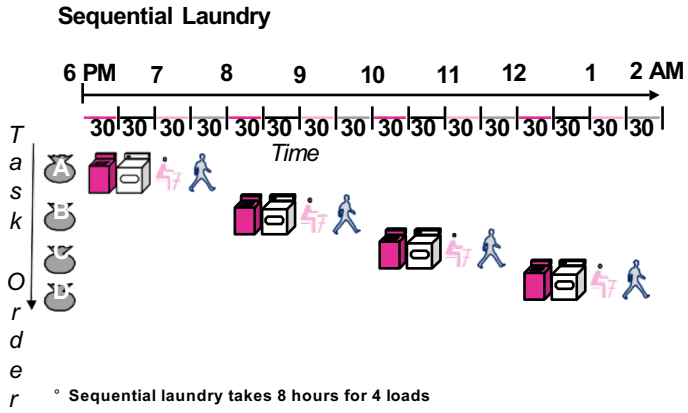
Folder takes 30-min

Stasher takes 30-min to put clothes in drawers

Computer Architecture & Assembly

Basic Pipelining

- By Example: Laundry pipelining



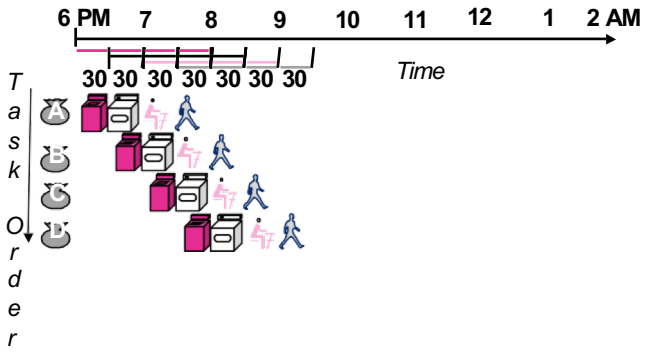
° Sequential laundry takes 8 hours for 4 loads

Computer Architecture & Assembly

Basic Pipelining

- By Example: Laundry pipelining

Pipelined Laundry: Start work ASAP



- **Pipelined laundry takes 3.5 hours for 4 loads!**

Computer Architecture & Assembly

Basic Pipelining

- By Example: Pipelining Lessons
  - Pipelining doesn't help latency of single task, it helps throughput of entire workload
  - Multiple tasks operating simultaneously using different resources
  - Potential speedup = Number of pipeline stages
  - Pipeline rate limited by slowest pipeline stage
  - Unbalanced lengths of pipeline stages reduces speedup
  - Time to "fill" pipeline and time to "drain" it reduces speedup
  - Stall for Dependences

Advanced Computer Architecture –Review

Basic Pipelining

- Design Issues:
  - The pipe stages are hooked together, thus all stages must be ready to proceed at the same time
  - The time required between moving an instruction one step down the pipeline is a processor cycle
  - Length of a processor cycle is determined by the time required for the slowest pipe stage
  - The pipeline designer's goal is to balance the length of each pipeline stage
  - If the stages are perfectly balanced, then the time per instruction on the pipelined processor—assuming ideal conditions—is equal to

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

Advanced Computer Architecture –Review

Basic Pipelining

- Design Issues:
  - Practically, time per instruction does NOT have its minimum possible value because:
    - The pipe stages are NOT perfectly Balanced!
    - Pipelining has its own OVERHEADs

Advanced Computer Architecture –Review

Basic Pipelining

- Single-cycle Vs Multi-cycle Machines
  - Single Cycle: Machines that process single instruction in one cycle
    - Cycle time has to be long enough to process slowest instruction
  - Multi Cycle: Machines that process single instruction in multiple cycles
    - Break the instruction into smaller steps (e.g., IF, ID, EX, MEM, WB)
    - Execute each step (instead of the entire instruction) in one cycle
    - Cycle time will be: time it takes to execute the longest step
    - Keep all the steps to have similar length

Advanced Computer Architecture –Review

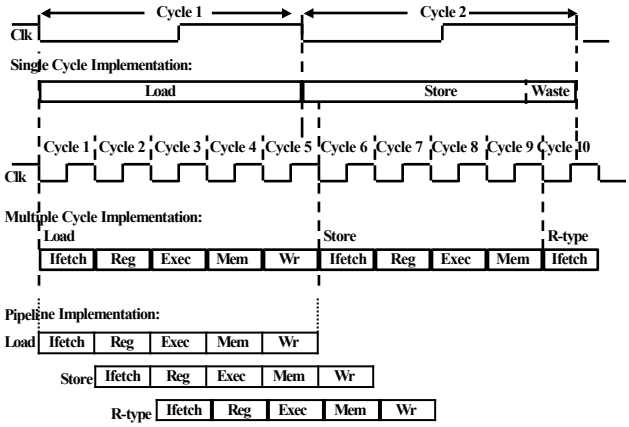
Basic Pipelining

- Single-cycle Vs Multi-cycle Machines
  - Example: Suppose we execute 100 instructions
  - Single Cycle Machine
    - 45ns/cycle x 1 CPI x 100 inst = 4500 ns
  - Multi-cycle Machine
    - 10 ns/cycle x 4.6 CPI (due to inst. mix) x 100 inst = 4600 ns
  - Ideal pipelined (multi-cycle) machine
    - 10ns/cycle x (1CPIx100inst+4cycle drain) =1040 ns

Advanced Computer Architecture –Review

Basic Pipelining

- Single-cycle, Multi-cycle, Pipelined Machines



Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Properties that benefit pipelining in RISC
    - All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register)
    - The only operations that affect memory are LOAD and STORE
    - The instruction formats are few in number, with all instructions typically being one size

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Typical 5-cycles of RISC instructions
    - Instruction Fetch (IF): Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding to the PC
    - Instruction Decode (ID): Decode the instruction and read the registers corresponding to register source specifiers from the register file
    - Execution (EX): The ALU operates on the operands prepared in the prior cycle, performing functions depending on the instruction type

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Typical 5-cycles of RISC instructions
    - Memory Access (MEM): For LOAD, the memory does a read using the effective address computed in the previous cycle. For STORE, the memory writes the data from the second register read from the register file using the effective address
    - Write Back (WB): Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction)

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Fetch –IF Stage

Program to Execute

```
0x100:   0x00b50533
0x104:   0x00d60633
0x108:   0x40c50533
   ⋮
```

Which instruction is executing?

The program counter (PC) tracks the current position in the program:

PC: 0x100 = 100_{16}

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Fetch –IF Stage

Program to Execute

```
0x100:   0x00b50533
0x104:   0x00d60633
0x108:   0x40c50533
   ⋮
```

What instruction is this?

Instruction: 0x00b50533 = $b50533_{16}$ = $0000000|01011|01010|000|01010|0110011_2$

add a0, a0, a1

next

PC: PC + 4 = 0x104

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Fetch –IF Stage

Program to Execute

```
0x100:   0x00b50533
0x104:   0x00d60633
0x108:   0x40c50533
   ⋮
```

What instruction is this?

Instruction: 0x00d60633 =
$0000000|01101|01100|000|01100|0110011_2$

add a2, a2, a3

next
PC: PC + 4 = 0x108

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Fetch –IF Stage

Program to Execute

```
0x100:   0x00b50533
0x104:   0x00d60633
0x108:   0x40c50533
   ⋮
```

What instruction is this?

Instruction: 0x40c50533 =
$0100000|01100|01010|000|01010|0110011_2$

sub a0, a0, a2

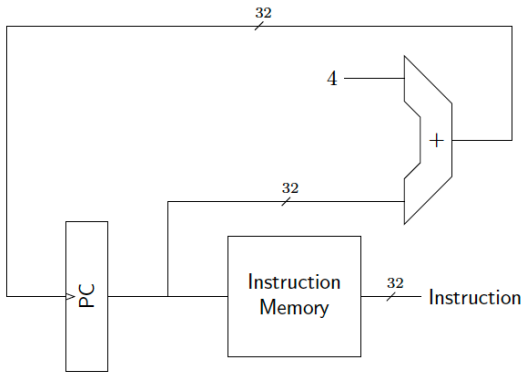
next
PC: PC + 4 = 0x10C

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Fetch –IF Stage

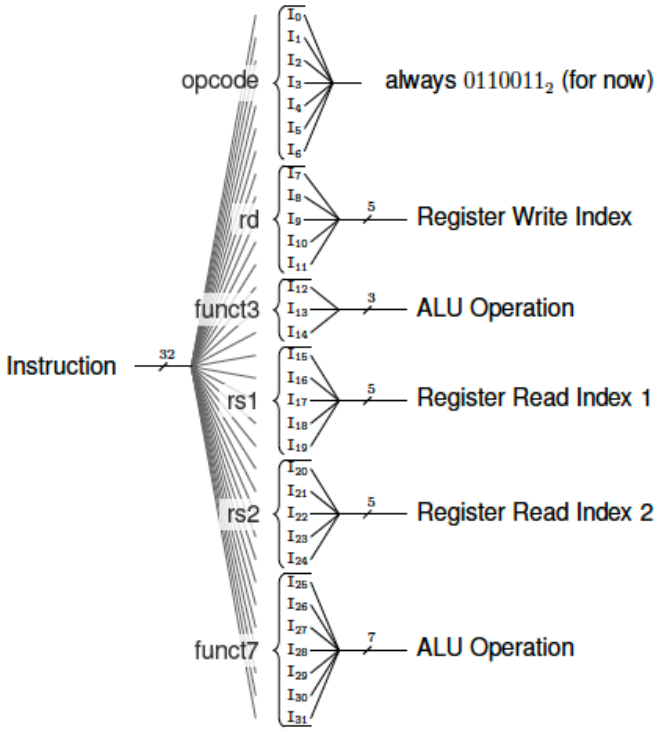
Instructions are stored in memory and loaded using the PC:



Advanced Computer Architecture –Review

Basic Pipelining

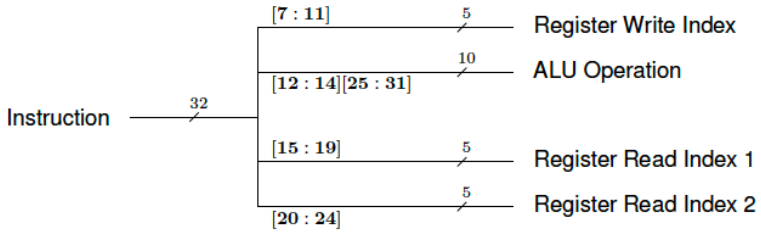
- Pipelining in RISC Architecture:
  - Instruction Decode –ID Stage
  - Instruction decoding (can be seen as) just a simple multiplexing of wires!



Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Decode –ID Stage
  - A more compact way of indicating multiplexing wires:



Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Execute –EXE Stage

Program to Execute	What is the instruction doing?
0x100: 0x00b50533	Instruction: **add** a0, a0, a1
0x104: 0x00d60633	
0x108: 0x40c50533	Read Registers: a0 and a1
⋮	Arithmetic: addition
	Write Register: a0

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Execute –EXE Stage

Program to Execute	What is the instruction doing?
0x100: 0x00b50533	Instruction: **add** a2, a2, a3
0x104: 0x00d60633	
0x108: 0x40c50533	Read Registers: a2 and a3
⋮	Arithmetic: addition
	Write Register: a2

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Execute –EXE Stage

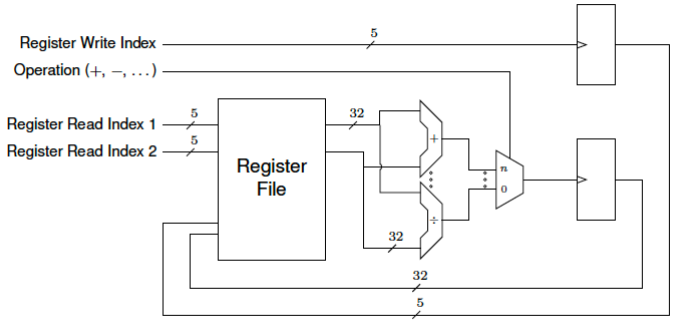
Program to Execute		What is the instruction doing?
0x100:	0x00b50533	Instruction: **sub** a0, a0, a2
0x104:	0x00d60633	
0x108:	0x40c50533	Read Registers: a0 and a2
	⋮	Arithmetic: subtraction
		Write Register: a0

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Execute –EXE Stage

Operands are read/written from/to a memory (register file):



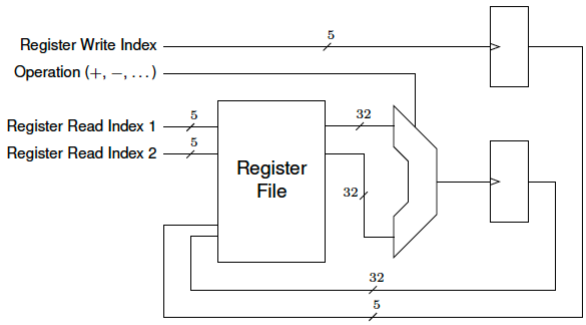
Register indices and operation depend on the executed instruction.[1]

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Instruction Execute –EXE Stage

Simplify by collapsing the Arithmetic/Logic Unit (ALU):

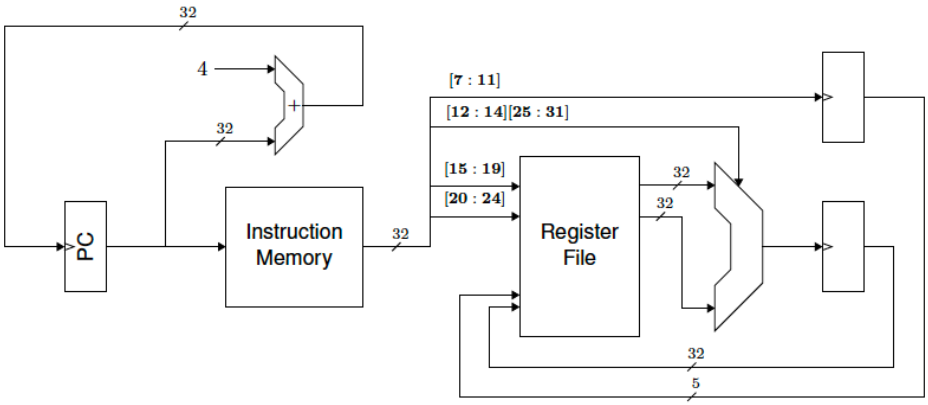


Register indices and operation depend on the executed instruction.[2]

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Putting it All Together



Advanced Computer Architecture –Review

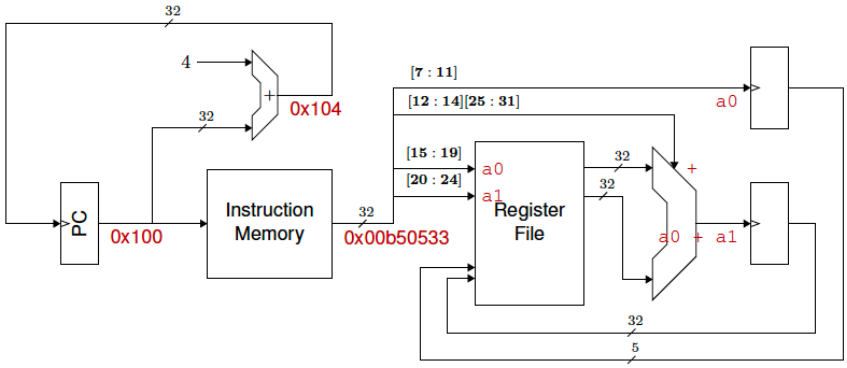
Basic Pipelining

- Pipelining in RISC Architecture:
  - Putting it All Together

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:

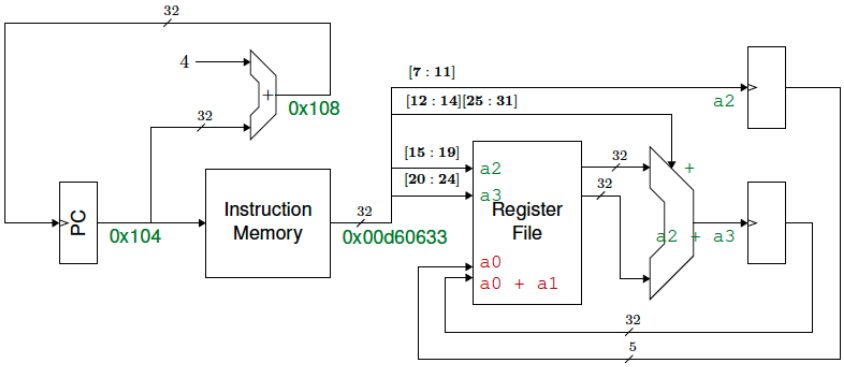



```
0x100:   0x00b50533   add a0, a0, a1
0x104:   0x00d60633   add a2, a2, a3
0x108:   0x40c50533   sub a0, a0, a2
```

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:

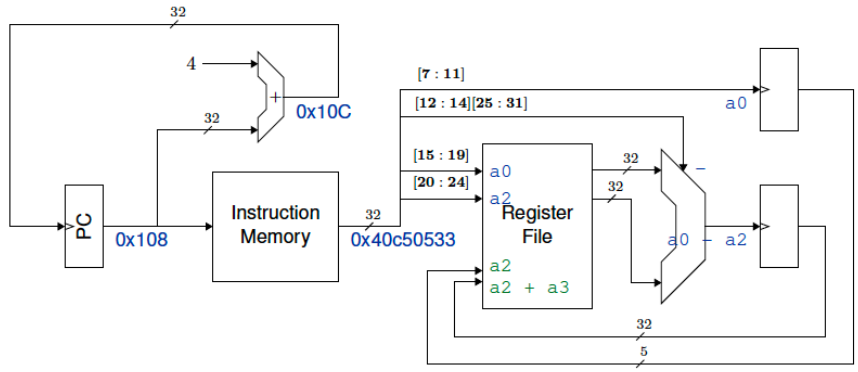



```
0x100:   0x00b50533   add a0, a0, a1
0x104:   0x00d60633   add a2, a2, a3
0x108:   0x40c50533   sub a0, a0, a2
```

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:




```
0x100:   0x00b50533   add a0, a0, a1
0x104:   0x00d60633   add a2, a2, a3
0x108:   0x40c50533   sub a0, a0, a2
```

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Classic 5-stage Pipeline of RISC Processor

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i + 1		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Its not as simple as it looks!
Lets see why

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture:
  - Two more issues to be resolved before we move to Pipeline Hazards!
  - How to process Immediate values?
  - How to update PC for branches?

RISC-V Instruction Classes and Formats

- **R-Format**:
  - Instructions operating on registers (thus the *R*)
  - Examples: **`add`**, **`or`**, **`sll`**, **`slt`**, **`div`**, . . .

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

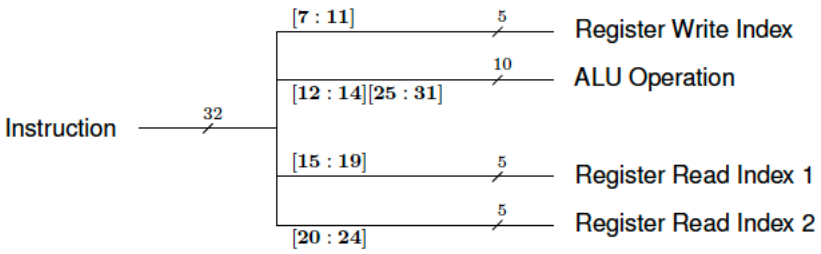
- **I-Format**:
  - Instructions taking an immediate constant (thus the *I*)
  - Examples: **`addi`**, **`andi`**, **`srai`**, **`sltiu`**, **`lw`**, **`jalr`**, . . .

imm	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

RISC-V Instruction Classes and Formats

- **R-Format**:
  - Instructions operating on registers (thus the *R*)
  - Examples: **`add`**, **`or`**, **`sll`**, **`slt`**, **`div`**, . . .

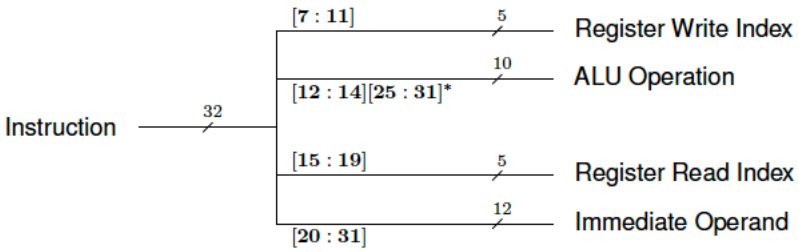
funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits



RISC-V Instruction Classes and Formats

- **I-Format**:
  - Instructions taking an immediate constant (thus the *I*)
  - Examples: **`addi`**, **`andi`**, **`srai`**, **`sltiu`**, **`lw`**, **`jalr`**, . . .

imm	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits



Advanced Computer Architecture –Review

Basic Pipelining

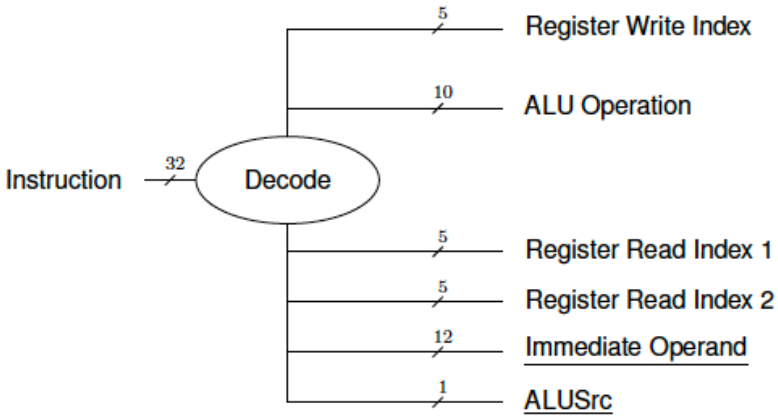
- Pipelining in RISC Architecture:

- Decoding is a block of combinatorial logic (we skip details)
- Decode stage issues additional control signals to following pipeline stages to process all Instruction Formats
- Operands of all instruction classes are extracted, since in all cases:
  - Read Registers 1 and 2 (rs1/rs2) can be accessed without side-effects
  - The ALU has to perform some operation
  - A new value is written to Write Register (rd)

Advanced Computer Architecture –Review

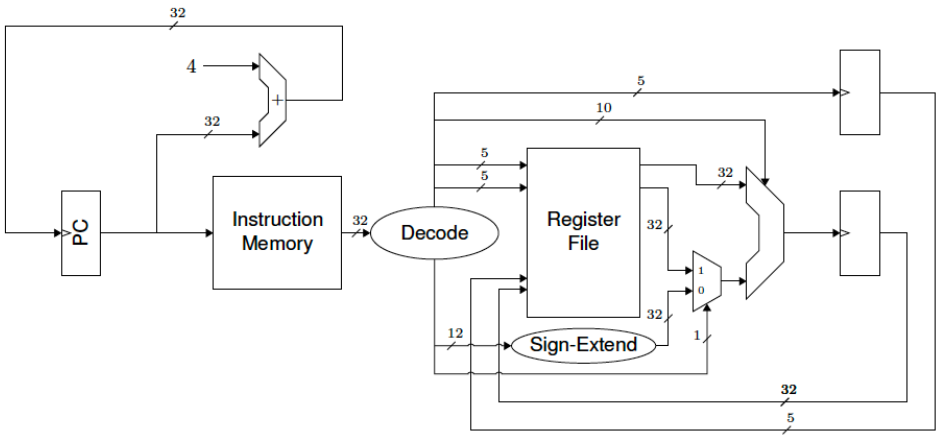
Basic Pipelining

- Pipelining in RISC Architecture:



Advanced Computer Architecture –Review

Basic Pipelining –Processing Immediate Values



New: Optionally select either the immediate or register operand as input to the ALU.

Advanced Computer Architecture –Review

Basic Pipelining –Handling PC for Branches

What does a conditional branch exactly do?

Checks whether a condition is true or false

- Register 1 is (not) equal to Register 2
- Register 1 is greater or equal than Register 2
- Register 1 is less than Register 2

Update the program counter if condition is true

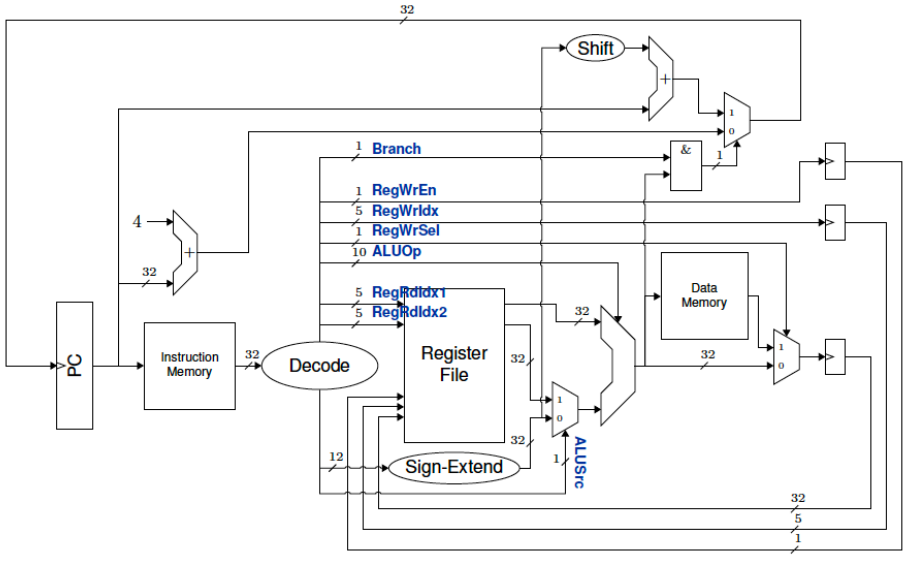
```
PC = PC + (immediate*4)= next instruction
```

Update the program counter if condition is false

```
PC = PC+4 = next instruction
```

Advanced Computer Architecture –Review

Basic Pipelining –Handling PC for Branches



Advanced Computer Architecture –Review

Basic Pipelining –Handling PC for Branches

Decode Signals

Instruction	Type	Branch	ALUSrc	ALUOp	RegRdIdx 1	RegRdIdx 2	RegWr Idx	RegWr En	RegWr Sel
add rd, rs1, rs2	R	0	1	+	rs1	rs2	rd	1	0
xor rd, rs1, rs2	R	0	1	⊕	rs1	rs2	rd	1	0
slt rd, rs1, rs2	R	0	1	<	rs1	rs2	rd	1	0
sll rd, rs1, rs2	R	0	1	*shl*	rs1	rs2	rd	1	0
addi rd, rs1, imm	I	0	0	+	rs1	/	rd	1	0
xori rd, rs1, imm	I	0	0	⊕	rs1	/	rd	1	0
lw rd, imm(rs1)	I	0	0	+	rs1	/	rd	1	1
bne rs1, rs2, imm	SB	1	1	≠	rs1	rs2	/	0	/

Advanced Computer Architecture –Review

Basic Pipelining

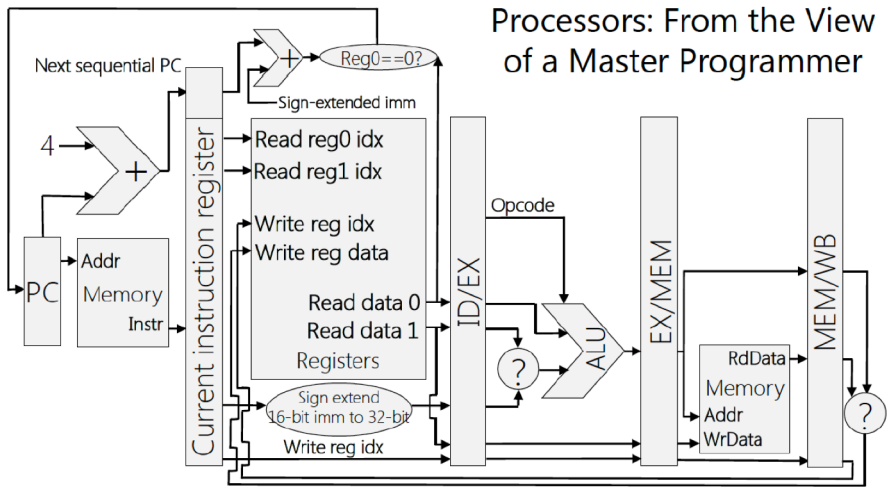
- Pipelining in RISC Architecture:
  - Classic 5-stage Pipeline of RISC Processor

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

56

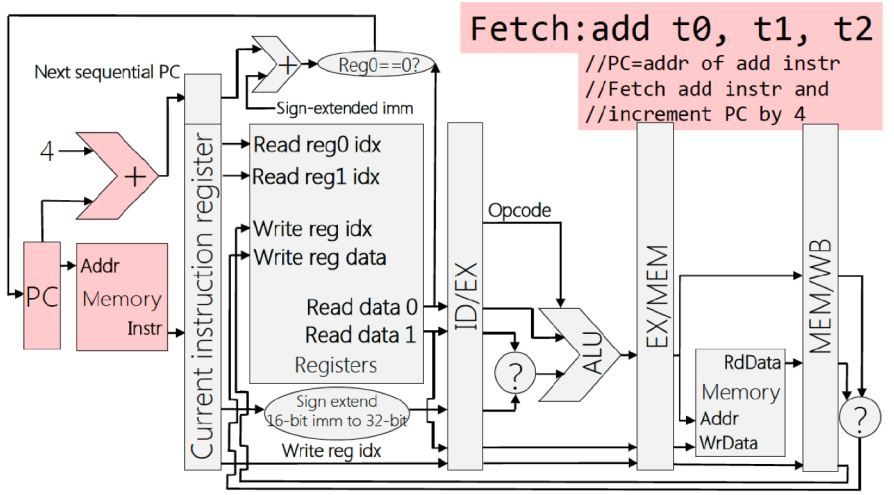
Computer Architecture

Basic Pipelining



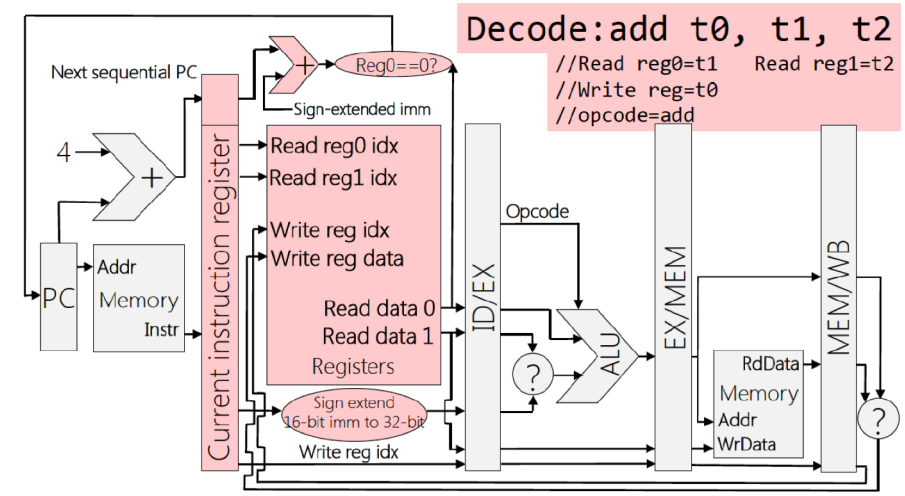
Computer Architecture

Basic Pipelining



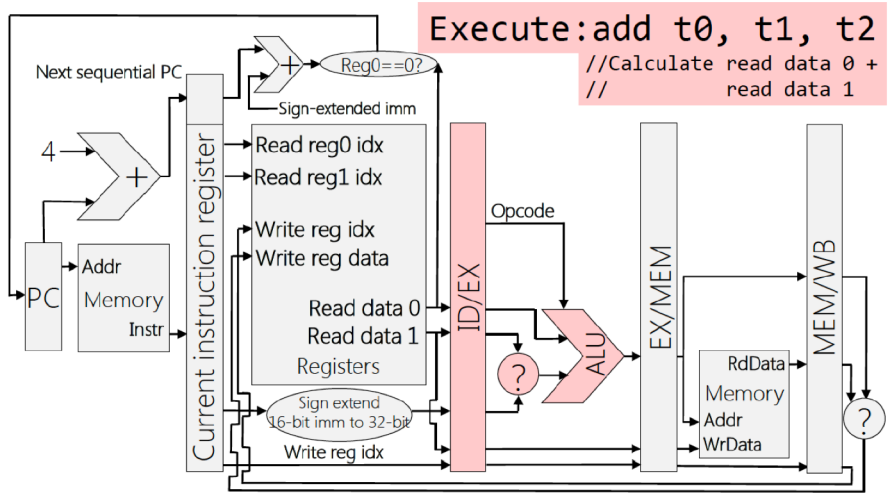
Computer Architecture

Basic Pipelining



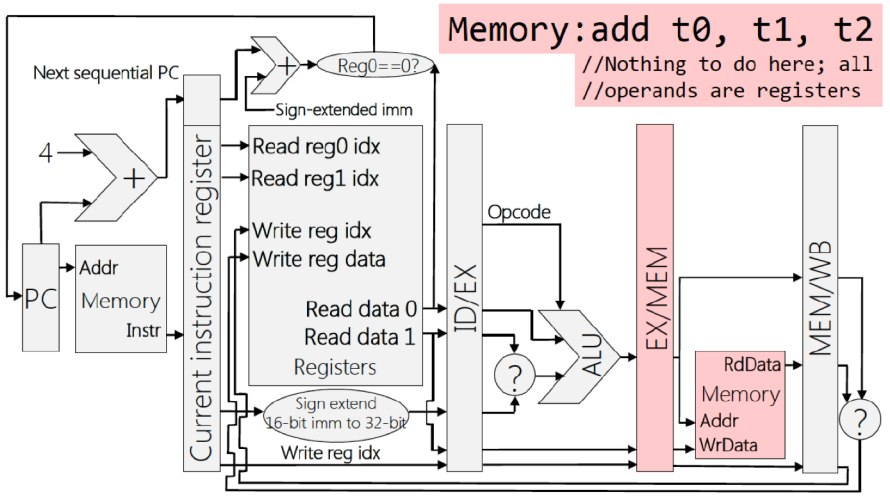
Computer Architecture

Basic Pipelining



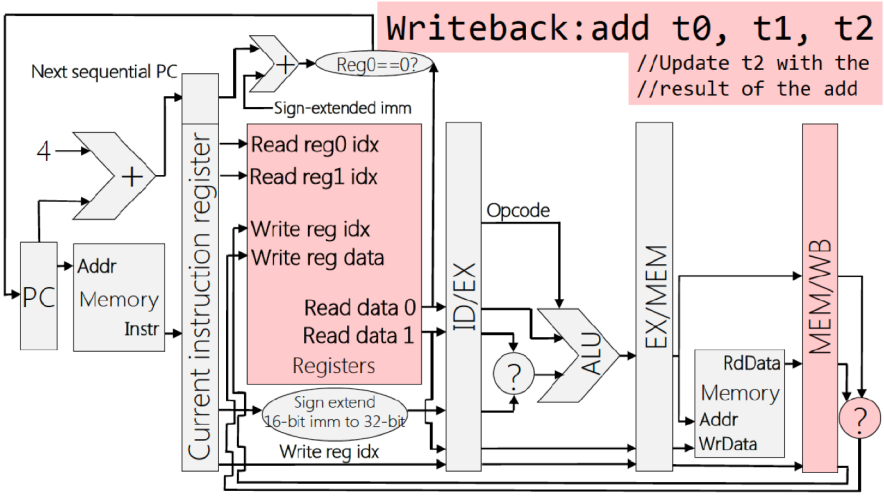
Computer Architecture

Basic Pipelining



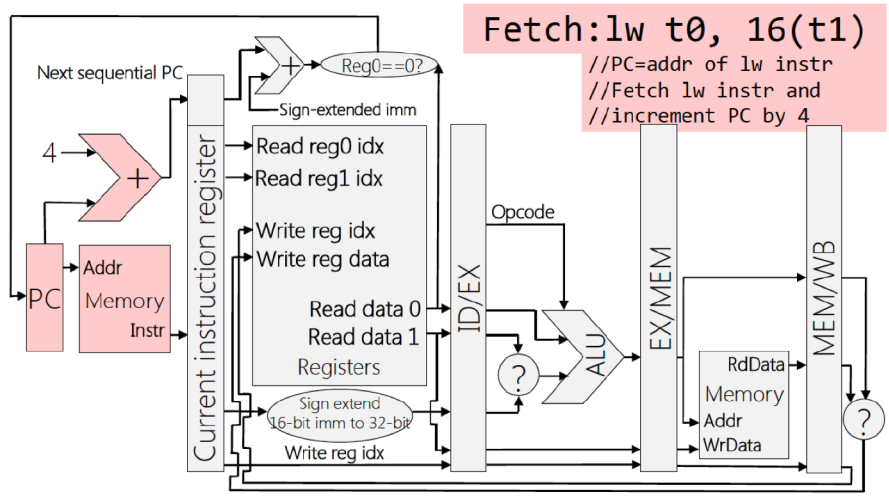
Computer Architecture

Basic Pipelining



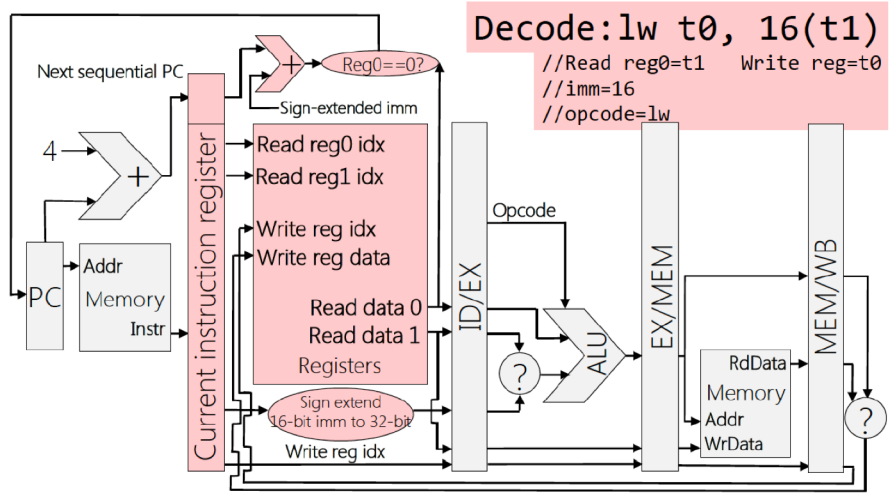
Computer Architecture

Basic Pipelining



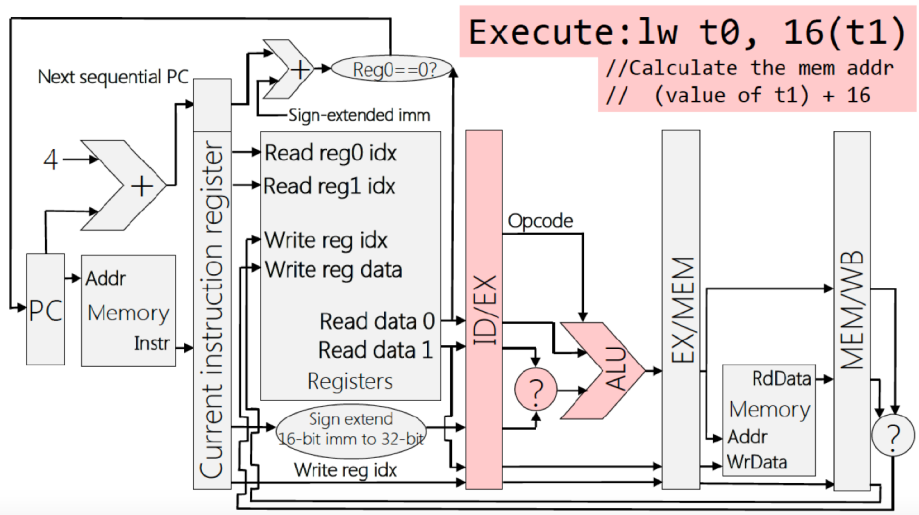
Computer Architecture

Basic Pipelining



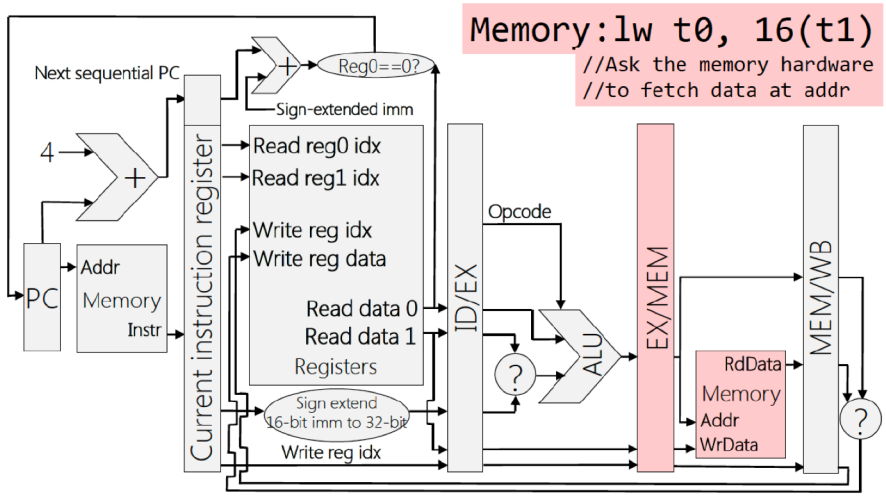
Computer Architecture

Basic Pipelining



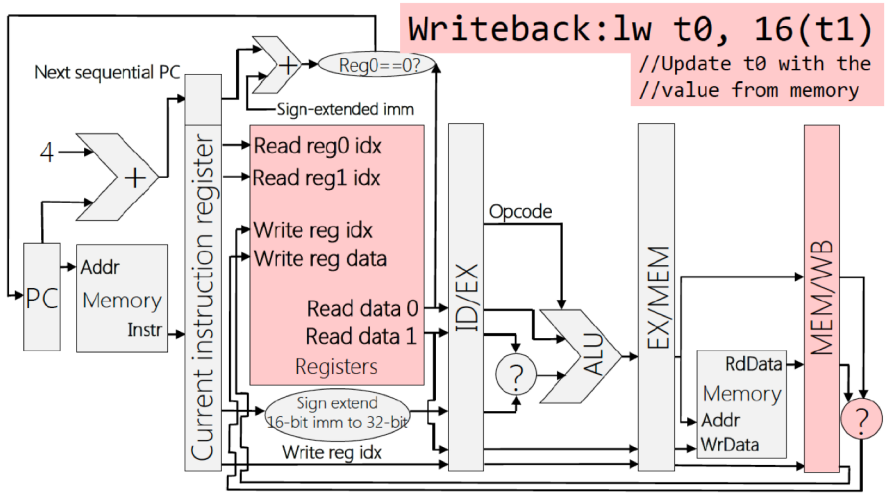
Computer Architecture

Basic Pipelining



Computer Architecture

Basic Pipelining



Advanced Computer Architecture –Review

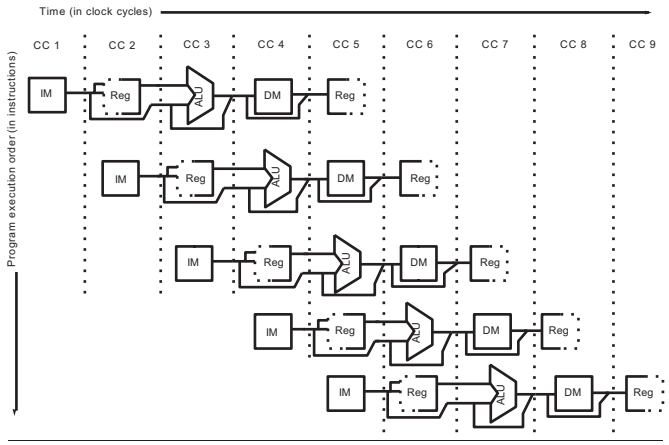
Basic Pipelining

- Pipelining in RISC Architecture:
  - What makes pipelining happen?
    - Availability of resources to perform different operations in different cycles
    - Ability to avoid performing different operation with the same data path resource on the same cycle!
    - Ability to ensure that the overlap of instructions in the pipeline cannot cause a conflict
      - Example: A single ALU cannot compute effective address and do addition operation in the same cycle

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture –Data path



Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture –Conflicts
  - I-cache and D-cache are accessed in the same cycle – it helps to implement them separately
  - Use of separate caches avoids conflict b/w instruction fetch & data memory access
    - Instruction memory (I-Cache)
    - Data memory (D-Cache)

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture -Conflicts
  - Register file can be used in multiple stages, for read and write operations
    - Distinct operations
    - Write in first half cycle
    - Read in the second half cycle

Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture -Conflicts
  - Instructions in different pipeline stage must not interfere with one another
    - Use of pipeline registers between successive stages
    - At the end of each cycle, results produced by each stage are stored in Registers
    - Results from registers are used as input to next stage

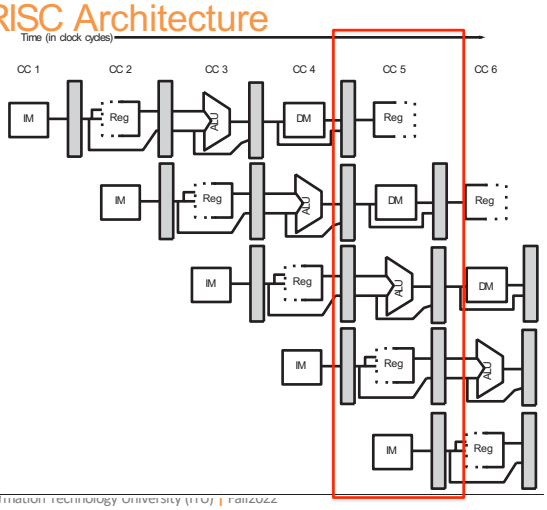
Advanced Computer Architecture –Review

Basic Pipelining

- Pipelining in RISC Architecture

Edge-triggered

IM: Instr. Mem
DM: Data Mem



Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - The program runs faster, while no single instruction runs faster!
    - Practically, instructions slightly extend execution time due to pipelining overheads
  - Issue 01: Pipeline Depth limitation
  - Issue 02: Pipeline imbalances
  - Issue 03: Pipeline overheads

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Issue 01: Pipeline Depth limitation
    - Due to execution time of individual instructions
  - Issue 02: Pipeline imbalances
    - Clock can not run faster than the time needed for slowest stage
  - Issue 03: Pipeline overheads
    - Pipeline registers add setup time (input stability time before the clock edge registers it)
    - Clock skew time (time between arrival of edge on two separate registers)
    - Upper limit on pipelining: CC = Clock Skew + Latching Overhead

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Example: Consider an unpipelined processor & assume that it has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. If these operations occur 40%, 20%, and 40%, respectively, and due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock, how much speedup in the instruction execution rate will we gain from a pipeline?

- Un-Pipelined case:

  Average instruction execution time = Clock cycle × Average CPI
  = 1 ns×[(40%+20%)×4+40%×5]
  = 1 ns × 4.4 = 4.4 ns

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - **Example:** Consider an unpipelined processor & assume that it has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. If these operations occur 40%, 20%, and 40%, respectively, and due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock, how much speedup in the instruction execution rate will we gain from a pipeline?

- Pipelined case:

  Average instruction execution time = Clock cycle × Average CPI
  = 1.2 ns×[(40%+20%+40%)×1]
  = 1.2 ns × 1 = 1.2 ns

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - **Example:** Consider an unpipelined processor & assume that it has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. If these operations occur 40%, 20%, and 40%, respectively, and due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock, how much speedup in the instruction execution rate will we gain from a pipeline?

- Pipelined case: Clock must run at the speed of the slowest stage plus

  overhead Average instruction execution time in this case = 1ns + 0.2 overhead

  = 1.2ns

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{4.4\text{ ns}}{1.2\text{ ns}} = 3.7 \text{ times}$$

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Pipeline works perfectly fine if instructions are independent –unfortunately, they are NOT!
  - Instructions inside a pipeline can depend on one another, which leads to Pipeline HAZARDS
  - Hazards prevent the next instruction in the instruction stream from executing during its *designated clock cycle*
  - Hazards reduce the performance from the ideal speedup gained by pipelining

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining –Hazards
  - . Structural Hazards
    - Arise from resource conflicts
    - Hardware cannot support all possible combinations of instructions simultaneously in overlapped execution
  - Data Hazards
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - Control Hazards
    - Arise from the pipelining of branches and other instructions that change the Program Counter

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining –Stalls
  - Pipeline Stalls
    - Pipeline stalls become a necessity to avoid Hazards
    - When an instruction is stalled, *all instructions issued later than the stalled instruction*—and hence not as far along in the pipeline—are also stalled
    - Instructions issued earlier than the stalled instruction, and hence farther along in the pipeline, must continue

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining –Stalls
  - Speedup effects of Stalls
    - Normal Speedup:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$
$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$
$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

    - Ideal CPI on a pipelined processor is almost always 1

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction
= 1 + Pipeline stall clock cycles per instruction

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining –Stalls
  - Speedup effects of Stalls
    - Ignoring the cycle time overhead of pipelining, assuming that the stages are perfectly balanced:

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining –Stalls
  - Speedup effects of Stalls
    - For the case where all instructions take the same number of cycles, which must also equal the number of pipeline stages

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

    - With no pipeline stalls, pipelining can improve performance equal to the depth of the pipeline!

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Structural Hazards
    - In a pipeline, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
    - If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a structural hazard.

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Structural Hazards
    - When a functional unit is not fully pipelined (not available for every instruction), a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle
    - When some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute
      - Example: A processor may have only one register-file write port, but the pipeline might want to perform two writes in a clock cycle

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Structural Hazards
    - Solution: Pipeline will stall one of the instructions until the required unit is available
    - Such stalls will increase the CPI from its usual ideal value of 1 under pipelining

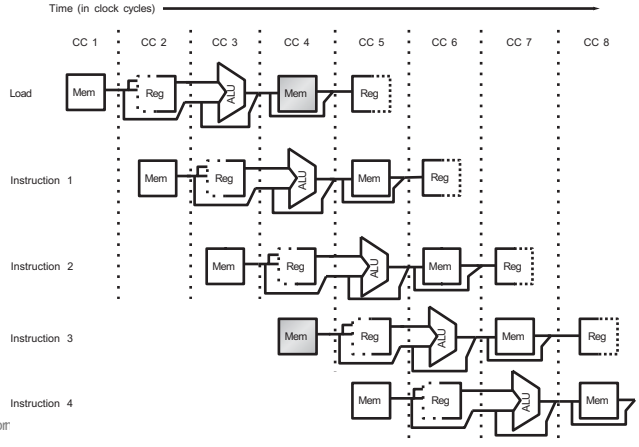
Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Structural Hazards

Example: Load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory

Stall = *pipeline bubble*



Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Structural Hazards

	Clock cycle number									
Instruction	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction i + 1		IF	ID	EX	MEM	WB				
Instruction i + 2			IF	ID	EX	MEM	WB			
Instruction i + 3				Stall	IF	ID	EX	MEM	WB	
Instruction i + 4						IF	ID	EX	MEM	WB
Instruction i + 5							IF	ID	EX	MEM
Instruction i + 6								IF	ID	EX

Stall in cycle 4 => causing not to complete any instruction in cycle 8

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Structural Hazards
    - If all other factors are equal, a processor without structural hazards will always have a lower CPI
    - Why would a designer allow structural hazards?

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards
    - A major effect of pipelining is to change the relative timing of instructions by overlapping their execution
    - Data hazards occur when the pipeline changes the order of read/write accesses to operands
    - Order differs from the order seen by sequentially executing instructions on an unpipelined processor

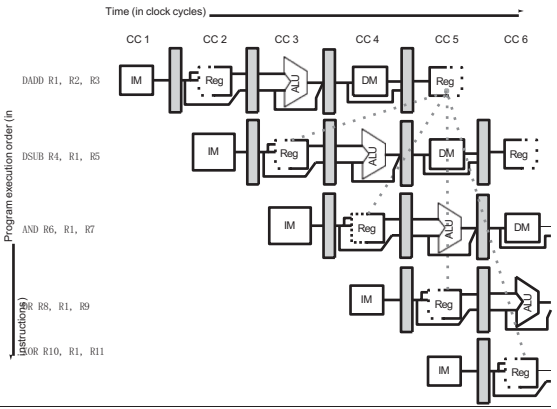
Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards –Identify

Sequential Instructions being pipelined

DADD	R1, R2, R3
DSUB	R4, R1, R5
AND	R6, R1, R7
OR	R8, R1, R9
XOR	R10, R1, R11



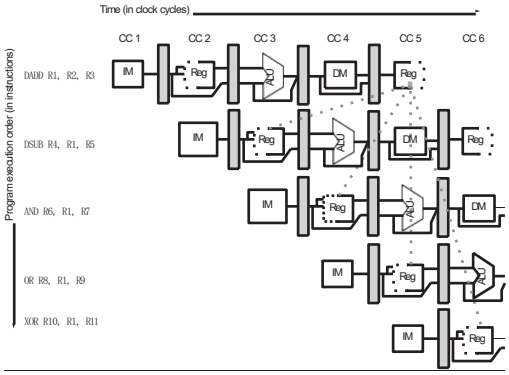
Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards –Identify

Sequential Instructions being pipelined

- What will happen to DSUB instr?
- All the instructions after the DADD use the result of the DADD instr. from R1
- DADD instruction writes the value of R1 in the WB pipe stage
- DSUB instruction reads the value during its ID stage.
- DSUB instruction will read the wrong value and try to use it.



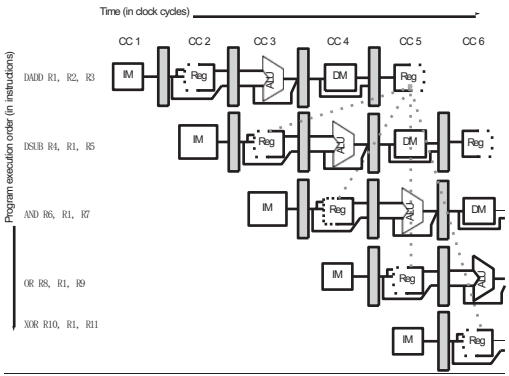
Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards –Identify

Sequential Instructions being pipelined

- The value used by the DSUB instruction is not even deterministic
- Logical assumption: DSUB would always use the value of R1 that was assigned by an instruction prior to DADD
- This is not always the case



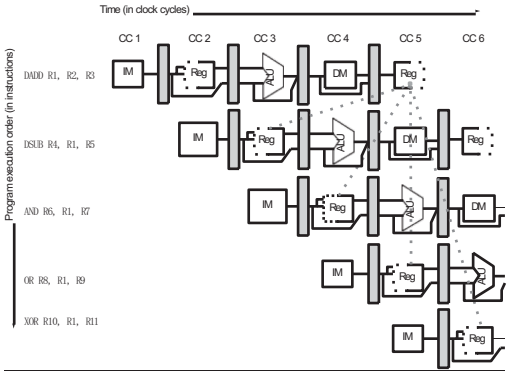
Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards –Identify

Sequential Instructions being pipelined

- What will happen to AND instruction? Affected or not?
- Write of R1 does not complete until the end of clock cycle 5.
- Thus, the AND instruction that reads the registers during clock cycle 4 will receive the wrong results



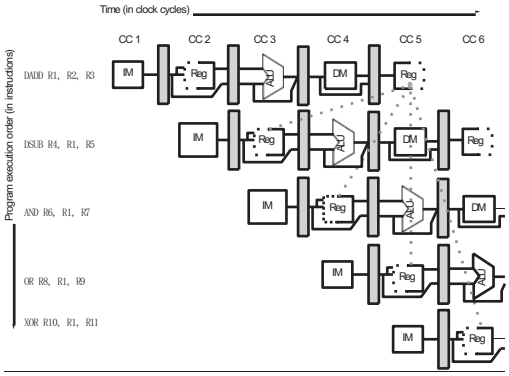
Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards –Identify

Sequential Instructions being pipelined

- What will happen to OR instruction? Affected or not?
- OR instruction operates without incurring a hazard
- Register file read is performed in the second half of the cycle and the write in the first half



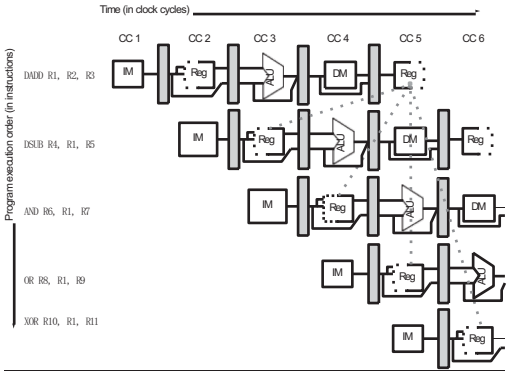
Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards –Identify

Sequential Instructions being pipelined

- What will happen to XOR instruction? Affected or not?
- XOR instruction operates without incurring a hazard
- Its register read occurs in clock cycle 6, after the register write
- READ After WRITE (RAW) Hazard



Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards – Preventive Measures
    - Hardware technique called Forwarding, Bypassing, or Short-Circuiting
    - Principle: Results by the subsequent instruction is NOT REALLY needed until after the precedent instruction produces it!
      - The DSUB doesn't need result until after the DADD actually produces it
    - If the result can be moved from the pipeline register where the DADD stores it to where the DSUB needs it, then the need for a stall can be avoided (sometimes, not always).

Advanced Computer Architecture –Review

Basic Pipelining

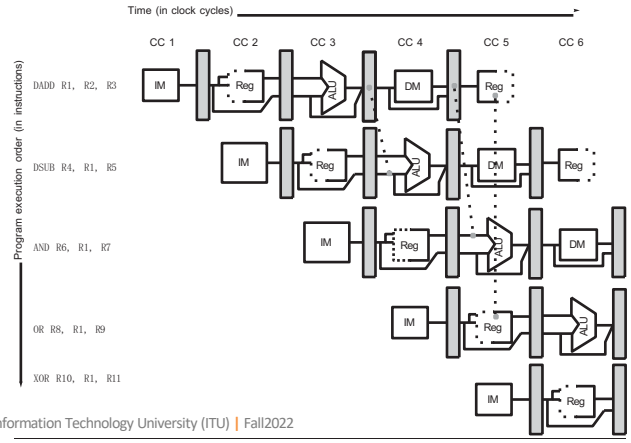
- Performance Issues in Pipelining
  - Data Hazards – Preventive Measures
    - Forwarding –How it works?
      - If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects (for processing) the forwarded result as the ALU input rather than the value read from the register file.

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Data Hazards – Preventive Measures

```
DADD    R1, R2, R3
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```



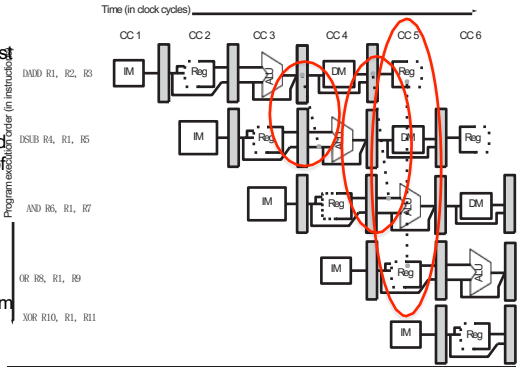
Advanced Computer Architecture –Review

Basic Pipelining

Performance Issues in Pipelining

Data Hazards – Preventive Measures

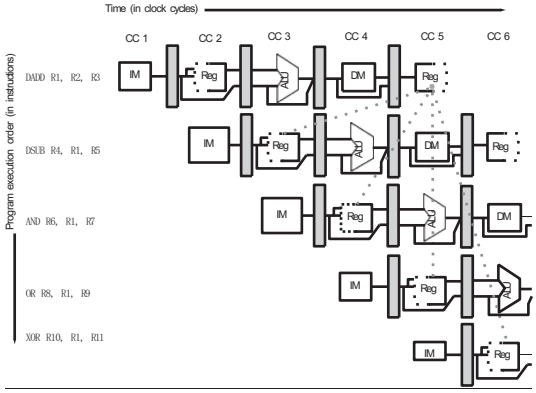
- Inputs for the DSUB & AND instructions forward from the pipeline registers to the first ALU input.
- OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half
- Forwarded result can go to either ALU input
- Both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers



Example-Without Forwarding

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11

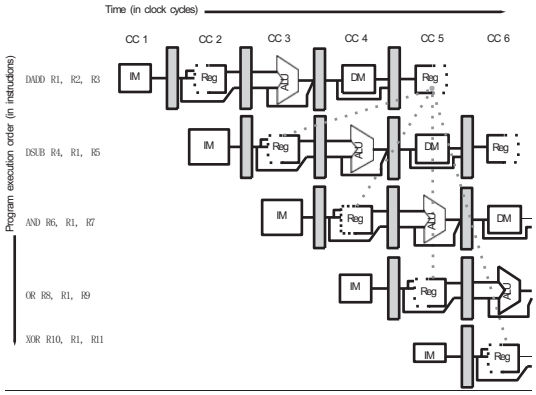


Instruc)ons	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC10
DADD R1, R2, R3											
DSUB R4, R1, R5											
AND R6, R1, R7											
OR R8, R1, R9											
XOR R10, R1, R11											

Example-Without Forwarding

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11

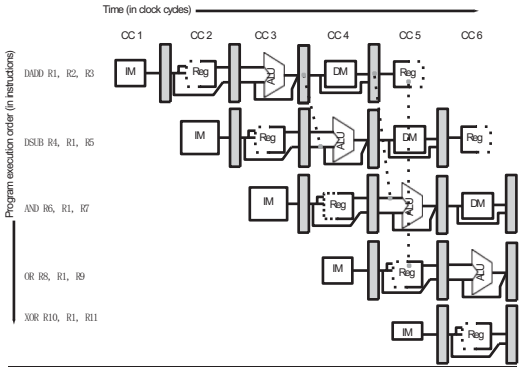


Instruc)ons	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
DADD R1, R2, R3	IF	ID	EX	MEM	WB						
DSUB R4, R1, R5		IF	stall	stall	ID	EX	MEM	WB			
AND R6, R1, R7			stall	stall	IF	ID	EX	MEM	WB		
OR R8, R1, R9			stall	stall		IF	ID	EX	MEM	WB	
XOR R10, R1, R11			Stall	stall			IF	ID	EX	MEM	WB

Example-With Forwarding

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11

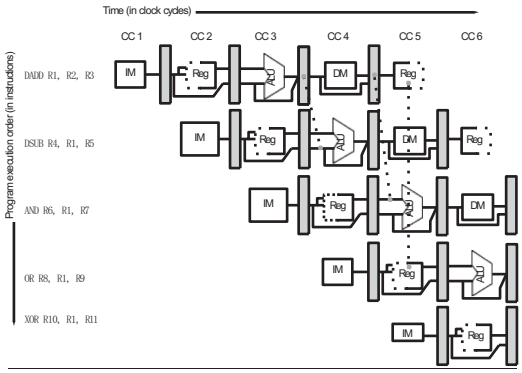


Instruc)ons	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC10
DADD R1, R2, R3											
DSUB R4, R1, R5											
AND R6, R1, R7											
OR R8, R1, R9											
XOR R10, R1, R11											

Example-With Forwarding

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

```
DADD    R1, R2, R3
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```



Instruc)ons	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC10
DADD R1, R2, R3	IF	ID	EX	MEM	WB						
DSUB R4, R1, R5		IF	ID	EX	MEM	WB					
AND R6, R1, R7			IF	ID	EX	MEM	WB				
OR R8, R1, R9				IF	ID	EX	MEM	WB			
XOR R10, R1, R11					IF	ID	EX	MEM	WB		

Advanced Computer Architecture –Review

Basic Pipelining

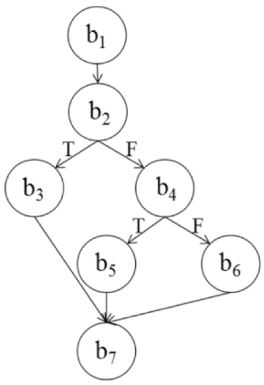
- Performance Issues in Pipelining
  - Control Hazards
    - Arise from branches and instructions that change PC
    - Performance losses are greater than data hazards. Why?
    - When a branch is executed, it may or may not change the PC to something other than its current value plus 4 (byte addressable)
      - If it does, instructions already fetched are useless operations

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards

```
S1;
if (cond S2)
   S3;
else if (cond S4)
   S5;
else S6;
S7;
```



Dr. Khurram Bhatti, Associate Professor | Information Technology University (ITU) | Fall2022 96

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Branch Delays
    - If a branch changes the PC to its target address, it is called a taken branch
    - If it falls through, it is not taken
    - If instruction *i* is a taken branch, then the PC is normally not changed until the end of ID, after the completion of the address calculation and comparison

Advanced Computer Architecture –Review

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Branch Delays
    - If instruction *i* is a taken branch
    - In ID stage, instruction *i* is known to be a branch
    - Fast equality test is used to compare the register values involved in the branch condition rather than a general comparator
    - The fact that it is an equality test is reflected in the ISA

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Preventive Measure
    - Simplest method of dealing with branches is to redo the fetch of the instruction following a branch
    - First IF cycle is essentially a stall: It never performs useful work
    - If the branch is Not Taken, the repetition of the IF stage is unnecessary
    - One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - How will the control unit know about Branch?
    - Prediction –Four simple compile time schemes

① Freeze or flush the pipeline
② Predicted-not-taken
③ Predicted-taken
④ Delayed branch

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties

① Freeze or flush the pipeline

- Holding or deleting any instructions after the branch until the branch destination is known
- Mainly attractive because of its simplicity both for hardware and software
- Penalty is fixed! Cannot be reduced through SW

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties

① Predicted-not-taken or Predicted- untaken

- Higher-performance, slightly more complex
- Treats every branch as not taken, simply allowing the hardware to continue as if the branch were not executed
- Implemented by continuing to fetch instructions as if the branch were a normal instruction
- If the branch is taken, need to turn the fetched instruction into a no-op and restart the fetch at the target address

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - ② Predicted-not-taken or Predicted- untaken

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - ③ Predicted-taken
      - Treats every branch as taken
      - No major advantage in this technique
      - The target address can not be known any time earlier than the branch outcome
      - Useful when the compiler can improve performance by organizing the code so that the most frequent path matches the hardware's choice, i.e., Predicted-taken!

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties

④ Delayed Branch

- After the branch instruction, ONE instruction is always executed whether or not the branch is taken
- This is called *Sequential Successor Instruction*

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

How to Fill Branch Delay Slot??

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties

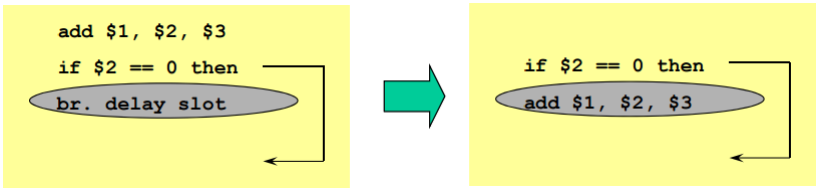
④ Delayed Branch

- From before (the branching instruction)
- From target (the branching instruction's target)
- From fall-through (of the branching instruction)
- From after (the branching instruction)

Advanced Computer Architecture

Basic Pipelining

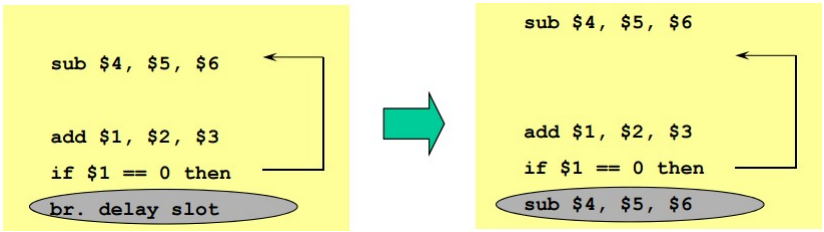
- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - ④ Delayed Branch- From before
      - – Best-Case Scenario!



Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - ④ Delayed Branch-From target
      - – Preferable when branch is highly predictable!



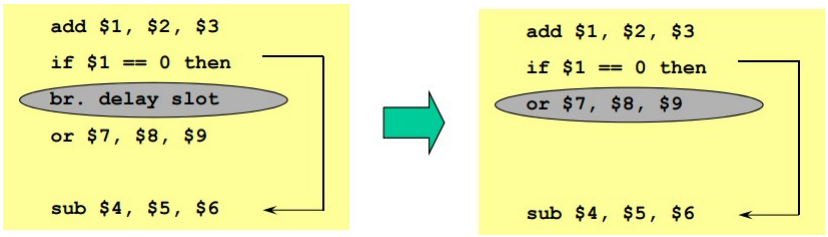
Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties

④ Delayed Branch-From fall-through

– The work will be wasted, but program must still execute correctly!



Advanced Computer Architecture

Basic Pipelining

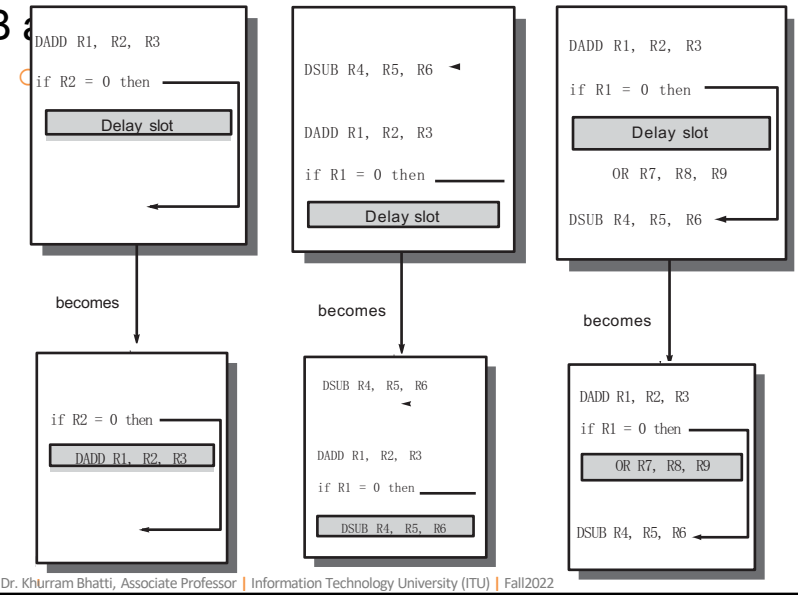
- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties

④ Delayed Branch- From after

Similar to “FROM BEFORE”, But select an independent instruction after the control/branch block.

Advanced Computer Architecture

B



Advanced Computer Architecture

Basic Pipelining

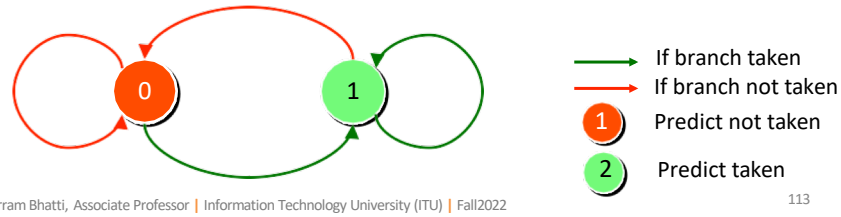
- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - Dynamic prediction: hardware decides the direction using dynamic information

① 1-bit Branch-Prediction Buffer
② 2-bit Branch-Prediction Buffer
③ Correlating Branch Prediction Buffer
④ Tournament Branch Predictor
⑤ Global Branch Predictor
⑥ …

Advanced Computer Architecture

Basic Pipelining

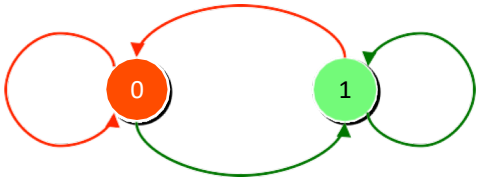
- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - Dynamic prediction: 1-bit predictor
      - Branch History Table
      - Use only one bit prediction
      - Accuracy is low



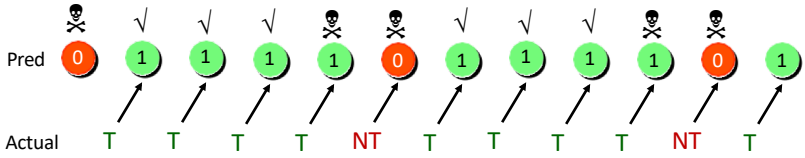
Advanced Computer Architecture

Basic Pipelining

- Dynamic prediction: 1-bit predictor



Ini;al State: Not taken

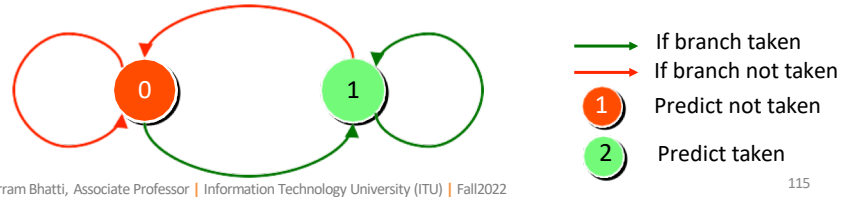


Aprox. 60% Accuracy

Advanced Computer Architecture

Basic Pipelining

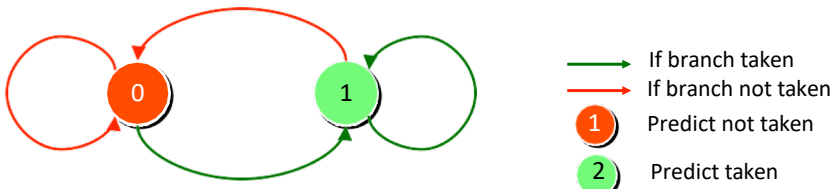
- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - 1-bit predictor –Exercise
    - Initial Prediction: T
    - Actual Branch-1: T-T-NT-T-T-NT-NT-NT-NT-T-NT-NT-T-T
    - Actual Branch-2: NT-T-NT-T-T-T-NT-T-NT-T-NT-NT



Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - 1-bit predictor –any issues??
      - Simple & efficient implementation
      - Changes mind fast!



Advanced Computer Architecture –Review

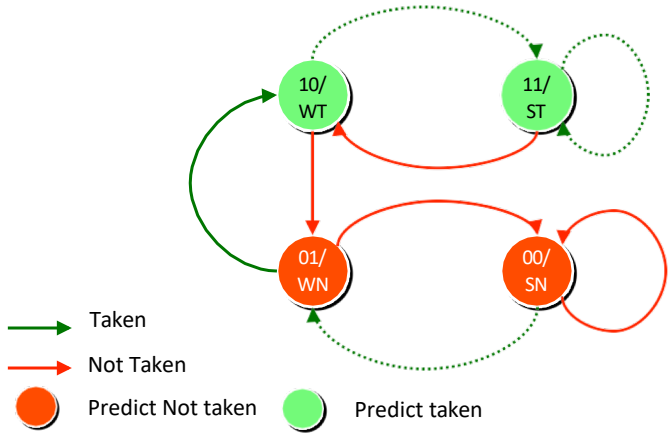
Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - Dynamic prediction: 2-bit predictor
      - Branch History Table
      - Changes prediction only if gets wrong prediction TWICE

Advanced Computer Architecture –Review

Basic Pipelining

- Dynamic prediction: 2-bit predictor



MSB: Direc;on bit
LSB: Hysteresis bit

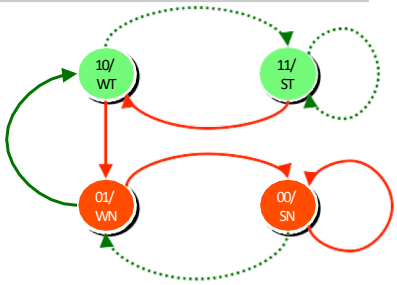
00/SN: Strongly Not Taken
01/WN: Weakly Not Taken
10/WT: Weakly Taken
11/ST: Strongly Taken

Advanced Computer Architecture –Review

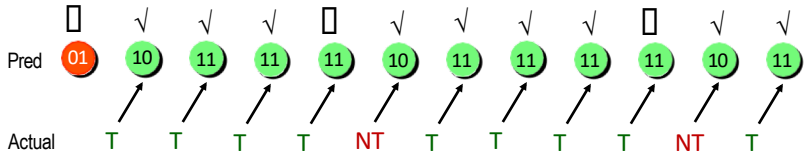
Basic Pipelining

- Dynamic prediction:

2-bit predictor



Ini;al State: Weakly not taken

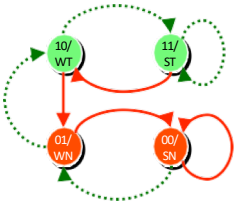


Approx. 75% Accuracy

Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - 2-bit predictor –Exercise
    - Initial Prediction: WT
    - Actual Branch-1: T-T-NT-T-T-NT-NT-NT-NT-T-NT-NT-T-T
    - Actual Branch-2: NT-T-NT-T-T-T-NT-T-NT-T-NT-NT



Advanced Computer Architecture

Basic Pipelining

- Performance Issues in Pipelining
  - Control Hazards –Reducing Pipeline Penalties
    - 2-bit predictor –Exercise
    - Initial Prediction: SN
    - Actual Branch-3: T-T-NT-NT-T-T-NT-NT-T-T-NT-NT-T-NT-NT-T-T
    - Actual Branch-4: NT-T-T-NT-T-T-T-NT-T-NT-T-NT-NT