

Introduction to Computer

略

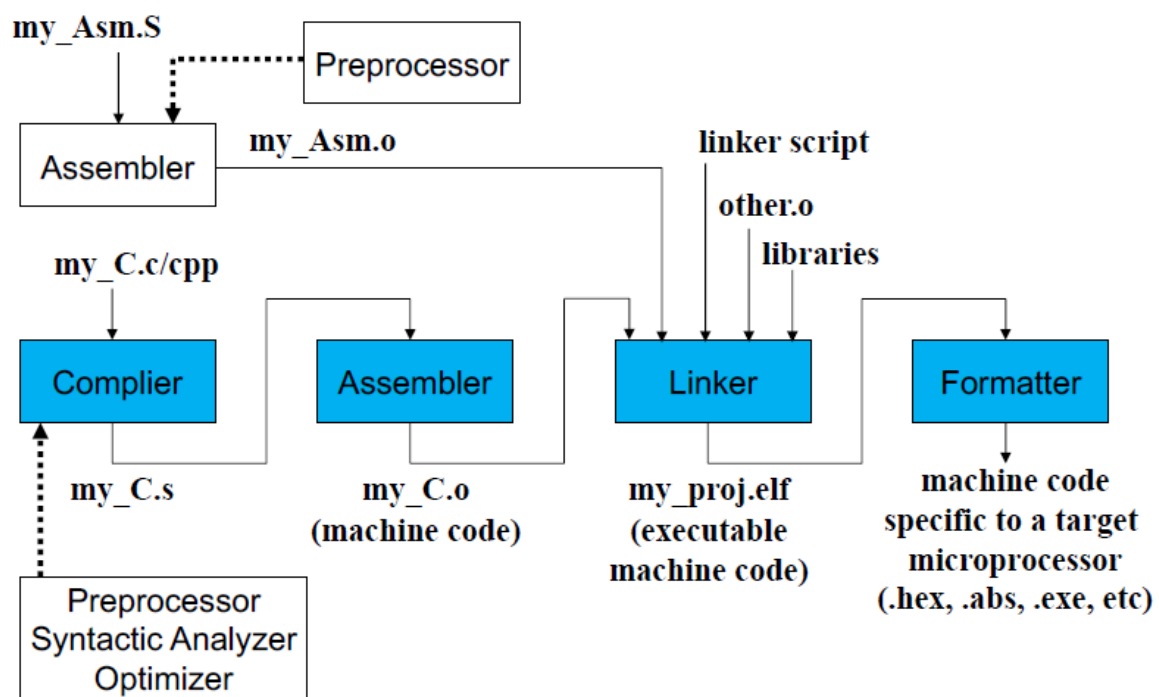
Assembly Programming

Operations and Operands

Levels of Program Code

- High-level language (translator: compiler)
- Assembly language (translator: assembler)
- Machine language

Processing Different Languages



High-level language -> **compiler** ->

Assembly language -> **assembler** ->

machine code -> **linker** (link with other related files) ->

executable machine code -> **formatter** ->

machine code specific to a target microprocessor

Assembly Language

Advantages:

- **Compilers** introduce **uncertainty** about execution time and size
- Use when **speed** and size **of** program are critical
- Can mix high-level language with assembly

Drawbacks:

- Can be very time consuming
- No assembler optimization
- Almost impossible to be portable
 - Different computers support different assembly languages that requires different assembler
 - Assembly languages are similar
- Hard to debug

Instruction Set (ISA)

- A collection of instructions that a computer understands.
- Different computers have different instruction sets
- Types:
 - Reduced Instruction Set Computer – RISC
 - Complex Instruction Set Computer – CISC

Arithmetic Operations

All arithmetic operations have three operands: **two sources** and **one destination**.

- operation destination source1 source2

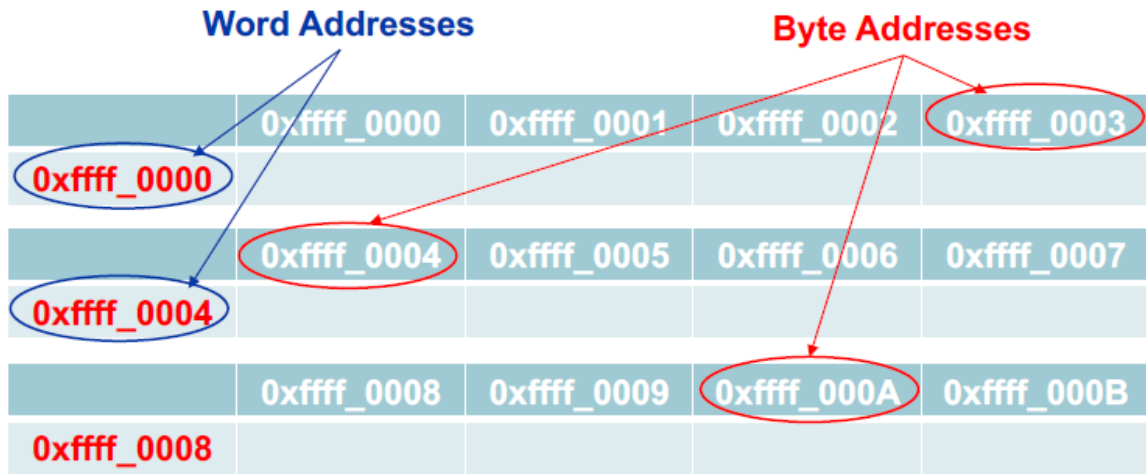
Operands in RISC-V Assembly

Register operands

- Arithmetic instructions use register operands
- RISC-V RV32 has a 32×32 -bit register file
 - x0: the constant value 0
 - x1 (ra): return address
 - x2 (sp): stack pointer
 - x3 (gp): global pointer
 - x4 (tp): thread pointer
 - x5 – x7, x28 – x31: temporaries
 - x8: frame pointer
 - x9, x18 – x27: saved registers
 - x10 – x11: function arguments/results
 - x12 – x17: function arguments

Memory operands

- Memory used mainly for composite data (Arrays, structures, dynamic data)
- Steps to use memory operands
 - **Load** values from memory **into registers**
 - Perform arithmetic **operations** with registers
 - **Store** result from register back to memory



- RISC-V memory is Little Endian
 - **Least-significant** byte at **smallest** byte address of a word
 - Big Endian: most-significant byte at smallest address

■ E.g.: 32-bit number 0x1020A0B0 (hexadecimal)

		0xffff_0000	0xffff_0001	0xffff_0002	0xffff_0003
Big Endian	0xffff_0000	10	20	A0	B0
		0xffff_0000	0xffff_0001	0xffff_0002	0xffff_0003
Little Endian	0xffff_0000	B0	A0	20	10

- Array Address: **Base Address + Offset**
 - e.g.: Address of A[8] = 0xFFEE0000 + (8 × 4)

Registers vs. Memory

- Registers are **faster** to access than memory
- Operating on memory data requires loads and stores (**More instructions**)
- Compiler must **use registers** for variables **as much as possible**

Immediate operands (constant)

- e.g.: addi x22, x22, 4

Other RISC-V Instructions

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

- Shift Operations
 - sll or slli: shift left logical
 - Fill vacated bits with 0 bits
 - sll by i bits = multiplies by 2^i
 - srl or srl: shift right logical
 - Fill vacated bits with 0 bits
 - srl by i bits = divides by 2^i (unsigned only)
 - srai: shift right arithmetic
 - Fill vacated bits with **sign bit**
- AND Operations
 - Useful to **mask bits** in a word
 - Select some bits, **clear others to 0**
 - e.g.: and x0, x10, x11

x11	0000 0000 0000 0000 0000 1101 1100 0000
x10	0000 0000 0000 0000 0011 1100 0000 0000
x9	0000 0000 0000 0000 0000 1100 0000 0000

- OR Operations
 - Useful to **include bits** in a word
 - Set some bits to 1** (selected part), leave others unchanged
 - e.g.: or x9, x10, x11

x11	0000 0000 0000 0000 0000 1101 1100 0000
x10	0000 0000 0000 0000 0011 1100 0000 0000
x9	0000 0000 0000 0000 0011 1101 1100 0000

- XOR Operations
 - Differencing operation
 - Set some bits to 1, leave others unchanged
 - e.g.: `xor x9,x10,x12 # NOT operation`

x10	00000000 00000000 00001101 11000000
x12	11111111 11111111 11111111 11111111
x9	11111111 11111111 11110010 00111111

Conditional Operations

- Each instruction can have a label (optional)
- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq`, `bne`, `blt`, `bge`, ...

Load upper immediate instruction

- `lui rd, constant`: Copies **20-bit constant** to bits [31:12] of `rd`
- For the occasional 32-bit constant, use `addi` after `lui`.

Signed and unsigned variants of instructions

- Bit 31 is sign bit
- Signed comparison: `blt`, `bge`
- Unsigned comparison: `bltu`, `bgeu`
- Sign extension will add sign bits before.

Function (Procedure) Call

Stored Program

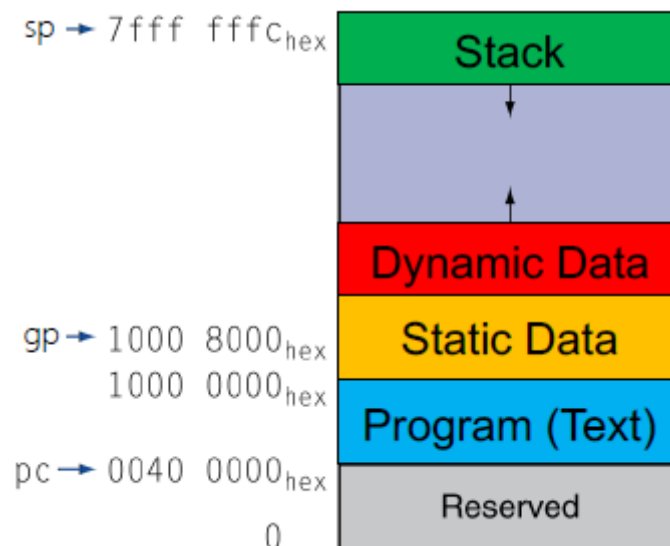
- Instructions are represented in binary, just like data
- Instructions and data are both stored in memory – stored program

Program Counter (PC)

- Each instruction is stored as a 32-bit word in program memory
 - has an address
 - when labeled, the label is equal to the address
- PC holds **address of an instruction** to be executed
- PC is a special register in CPU

Memory Layout

- **Text:** program code
- **Static data:** global/static variables
 - x3 (gp) initialized to the middle of this segment, 0x10008000 allowing \pm offset
- **Dynamic data:** heap
- **Stack:** storage for temporary variable in functions
 - x2 (sp) initialized to 0x7ffffffc, growing towards low address



Function Calling

- Steps for function calling operation
 1. **Place parameters** in registers x10 to x17 (function arguments)
 2. Transfer control to procedure
 3. Acquire storage on stack for procedure
 4. **Perform** procedure's operations
 5. **Place result** in register x10 and x11 for caller
 6. **Return** to place of call (address in x1)

Function Call Instructions

Function call: jump and link

```
jal x1, ProcedureLabel
```

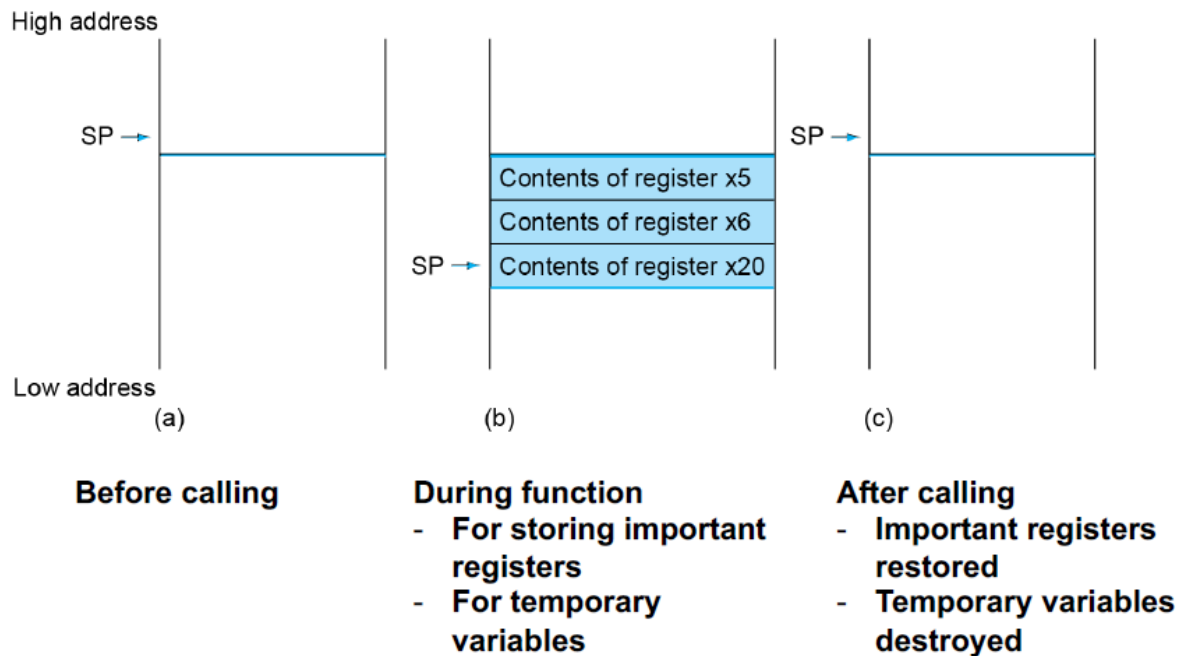
- **x1** \leq **PC + 4**, x1 is called return address reg.
- **PC** \leq **ProcedureLabel**

Function return: jump and link register

```
jalr x0, offset(x1)
```

- $x0 \leq PC + 4$ ($x0 \neq 0$, nothing happens)
- **PC \leq offset + return address stored in x1**, offset usually is 0 for function return
- Can also be used for computed jumps

Uses of Stack in Function Call



Leaf Function

- Functions that don't call other functions.
 - e.g.: (string copy)



Non-Leaf Functions

- Functions that call other functions.
- For nested call, caller needs to **save on the stack** before calling another function:
 - Its **return address**
 - Any **argument registers**
 - **Temporary registers needed** after the call
- **Restore from the stack** after the call

fact:

addi sp,sp,-8	Save return address and n on stack
sw x1,4(sp)	
sw x10,0(sp)	
addi x5,x10,-1	$x5 = n - 1$
bge x5,x0,L1	if $n \geq 1$, go to L1
addi x10,x0,1	Else, set return value to 1
addi sp,sp,8	Pop stack, don't bother restoring values
jalr x0,0(x1)	Return
L1: addi x10,x10,-1	$n = n - 1$
jal x1,fact	call fact($n-1$)
addi x6,x10,0	move result of fact($n - 1$) to x6
lw x10,0(sp)	Restore caller's n
lw x1,4(sp)	Restore caller's return address
addi sp,sp,8	Pop stack
mul x10,x10,x6	return $n * \text{fact}(n-1)$
jalr x0,0(x1)	return

Instruction Encoding


Representing Instructions

- 6 types (format):
 - R-type (Register)
 - I-type (Immediate)
 - S-type (Store)
 - U-type (Load upper immediate)
 - B-type (Branch), a.k.a. SB-type
 - J-type (Jump), a.k.a. UJ-type

Type	Field					
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
R-type	funct7	rs2	rs1	funct3	rd	opcode
I-type	immediate[11:0]		rs1	funct3	rd	opcode
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
B-type	immed[11,9:4]	rs2	rs1	funct3	immed[3:0,10]	opcode
U-type	immediate[19:0]				rd	opcode
J-type	immediate[19,9:0,10,18:11]				rd	opcode

R-type

add x9,x20,x21




funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_2 = 015A04B3_{16}$

I-type

addi x9, x20, 4



Immediate[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
4	20	0	9	19
000000000100	10100	000	01001	0010011

$00000000010010100000010010010011_2 = 004A0493_{16}$

lw x9, -4(x20)

Immediate[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
-4	20	2	9	3
111111111100	10100	010	01001	0000011

$11111111110010100010010010000011_2 = \text{FFCA2483}_{16}$

S-type

sw x9, -4(x20)

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

-4 = **1111111**_**11100**

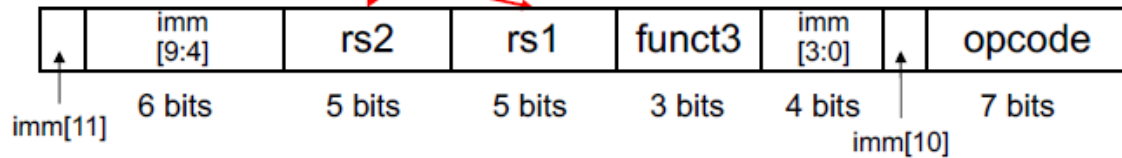
1111111	01001	10100	010	11100	0100011
-1	9	20	2	-4	35

$11111110100110100010111000100011_2 = \text{FE9A2E23}_{16}$

B-type

Branch Target address (Target PC) = **Current PC** + **immediate** × 2

beq x20, x21, Target



immediate = (Branch Target – Current PC) >> 1

e.g. immediate = -4 = **1_1_11111_1100**

1	111111	10101	10100	000	1100	1	1100011
----------	---------------	-------	-------	-----	-------------	----------	---------

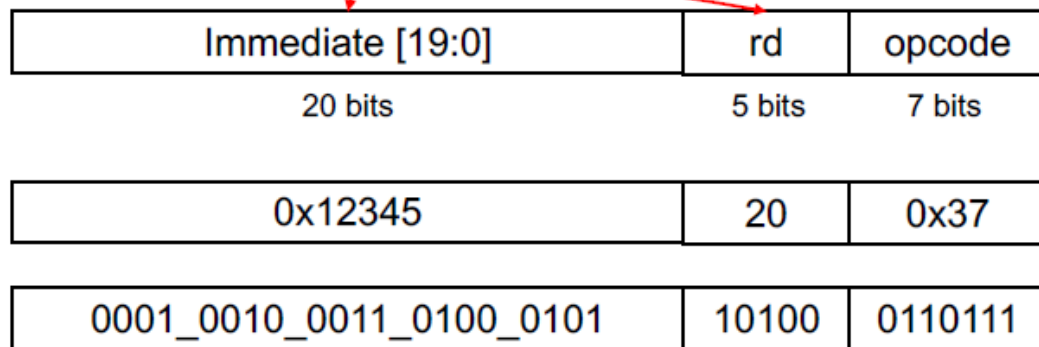
-1	21	20	0	25	99
-----------	----	----	---	-----------	----

$11111111010110100000110011100011_2 = \text{FF5A0CE3}_{16}$

U-type

For lui and auipc.

lui x20, 0x12345



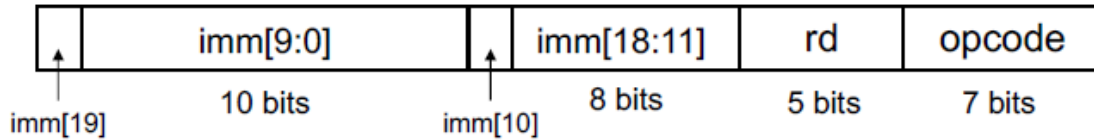
$00010010001101000101101000110111_2 = \text{12345A37}_{16}$

J-type

For jal:

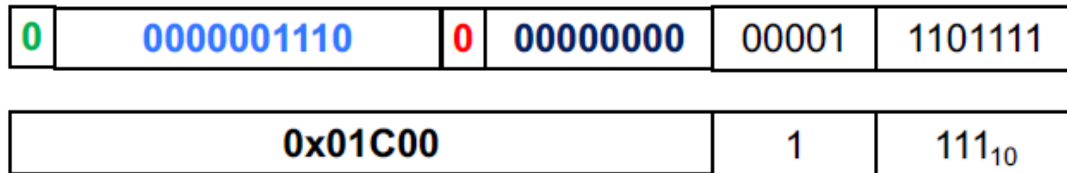
- $x1 \leq \text{PC} + 4$, x1 is called return address reg.
- **Target PC** \leq **Current PC** + **immediate** \times 2

```
jal x1, Target
```



immediate = (Target PC – Current PC) >> 1

e.g. immediate = 14 = 0_00000000_0_0000001110

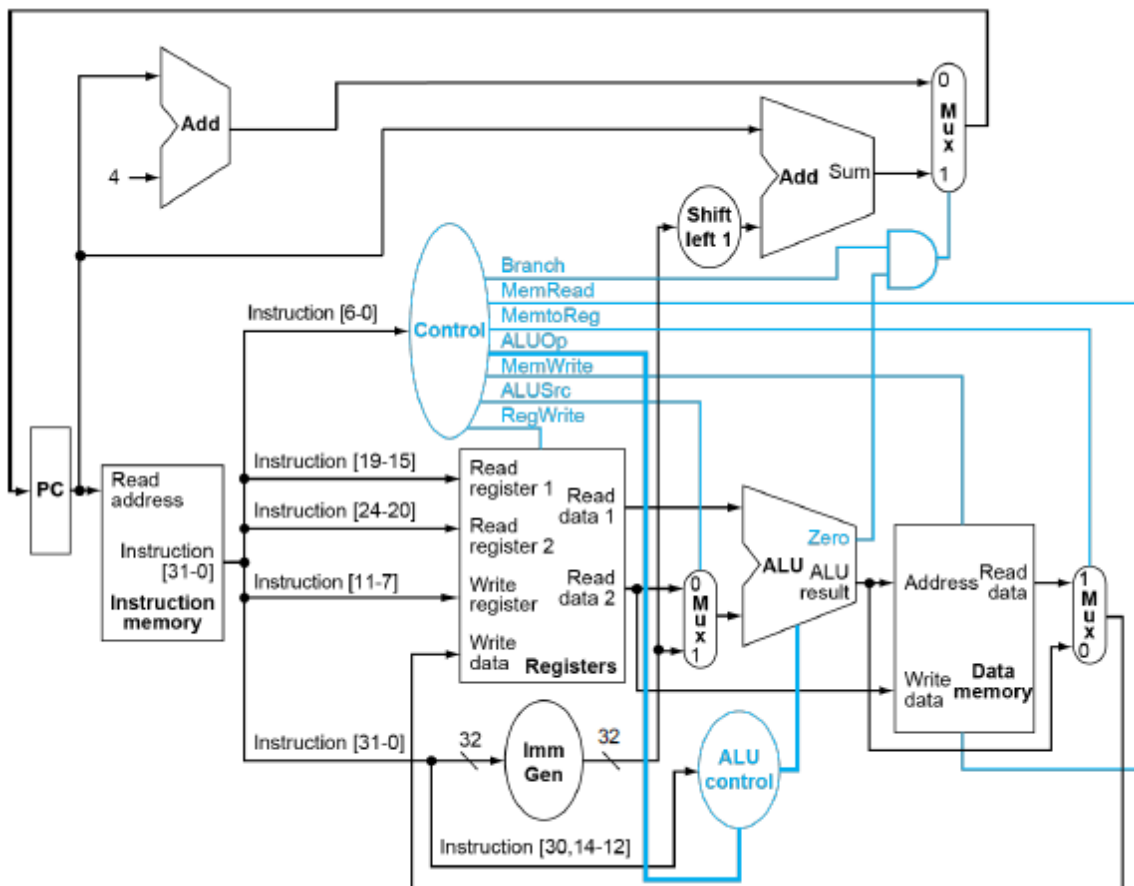


$$0000000111000000000000011101111_2 = 01C000EF_{16}$$

Performance Considerations

- Immediate bits are swirled around. This will:
 - Create difficulty for assemblers
 - But save hardware (muxes) on the critical path

Single Cycle Processor



ALU Control Unit

Instr.	Operation	ALU function	Opcode field	ALUOp (input)	i[30] (input)	funct3 (input)	ALU control (output)
lw	load register	add	XXXXXXX	00	X	010	0010
sw	store register	add	XXXXXXX	00	X	010	0010
beq	branch on equal	subtract	XXXXXXX	01	X	000	0110
R-type	add	add	100000	10	0	000	0010
	subtract	subtract	100010		1	000	0110
	AND	AND	100100		0	111	0000
	OR	OR	100101		0	110	0001

ALU control Signal	Function
0000	AND
0001	OR
0010	add
0110	subtract

Control Unit

Give different control signals when getting different opcode.

ImmGen

Generate immediate and **extend it to 32 bits**.

Performance Related Factors

- Algorithm
 - Determines number of operations executed
- Programming language, compiler, architecture
 - Determine number of machine instructions (lines of source code) executed per operation
- Processor and memory system
 - Determine how fast instructions are executed
- I/O system (including OS)
 - Determines how fast I/O operations are executed

Measure Computer Performance -- Execution Time

Elapsed time – for System Performance

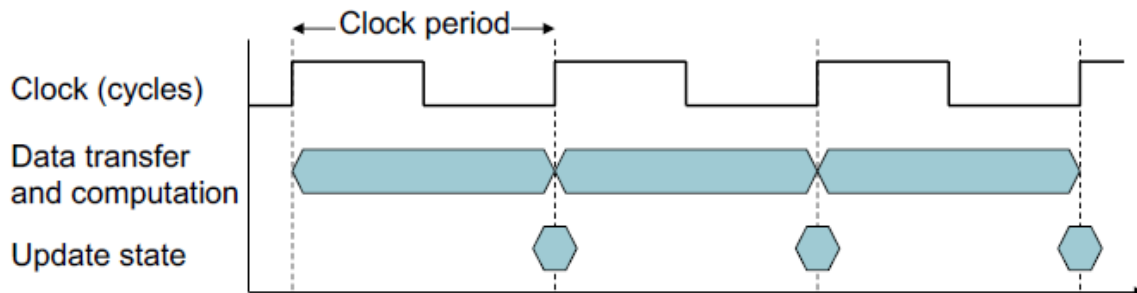
- Total execution time to complete a task, including: Processing, I/O, OS overhead, idle time, everything
- But doesn't completely reflect computer's performance if it focuses on better throughput

CPU time – for CPU Performance

- CPU execution time processing a task
 - Exclude I/O time, time spent for other tasks
- Comprises user CPU time and system CPU time because they are hard to differentiate
- Different programs run with different CPU performance and system performance

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: **duration** of a clock cycle, e.g., $250 \text{ ps} = 0.25 \text{ ns} = 250 \times 10^{-12} \text{ s}$
- Clock frequency (rate): **cycles per second**, e.g., $4.0 \text{ GHz} = 4000 \text{ MHz} = 4.0 \times 10^9 \text{ Hz}$

CPU Time

- Formula:
 - $\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time} = \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$
- Performance improved by
 - Reducing number of clock cycles
 - Increasing clock rate
 - Hardware designer must often **trade off clock rate against cycle count**

■ Computer A: 2GHz clock, 10s CPU time

■ Designing Computer B

- Aim for 6s CPU time
- Can do faster clock, but causes $1.2 \times$ clock cycles

■ How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\begin{aligned} \text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A \\ &= 10\text{s} \times 2\text{GHz} = 20 \times 10^9 \end{aligned}$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

- Clock Cycles = Instruction Count (IC) \times Cycles per Instruction (CPI)
 - CPU Time = IC \times CPI \times Clock Cycle Time = $\frac{IC \times CPI}{\text{Clock Rate}}$
 - Example:
- Computer A: Cycle Time = 250ps, CPI = 2.0
 - Computer B: Cycle Time = 500ps, CPI = 1.2
 - Same ISA
 - Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= 1 \times 2.0 \times 250\text{ps} = 1 \times 500\text{ps} \quad \leftarrow \text{A is faster...} \\ \text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= 1 \times 1.2 \times 500\text{ps} = 1 \times 600\text{ps} \\ \frac{\text{CPU Time}_B}{\text{CPU Time}_A} &= \frac{1 \times 600\text{ps}}{1 \times 500\text{ps}} = 1.2 \quad \leftarrow \text{...by this much}\end{aligned}$$

Summary

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Algorithm: affects IC, possibly CPI
- Programming language: affects IC, CPI
- Compiler: affects IC, CPI
- Instruction set architecture: affects IC, CPI, T_c (clock period)

Clocking Methodology

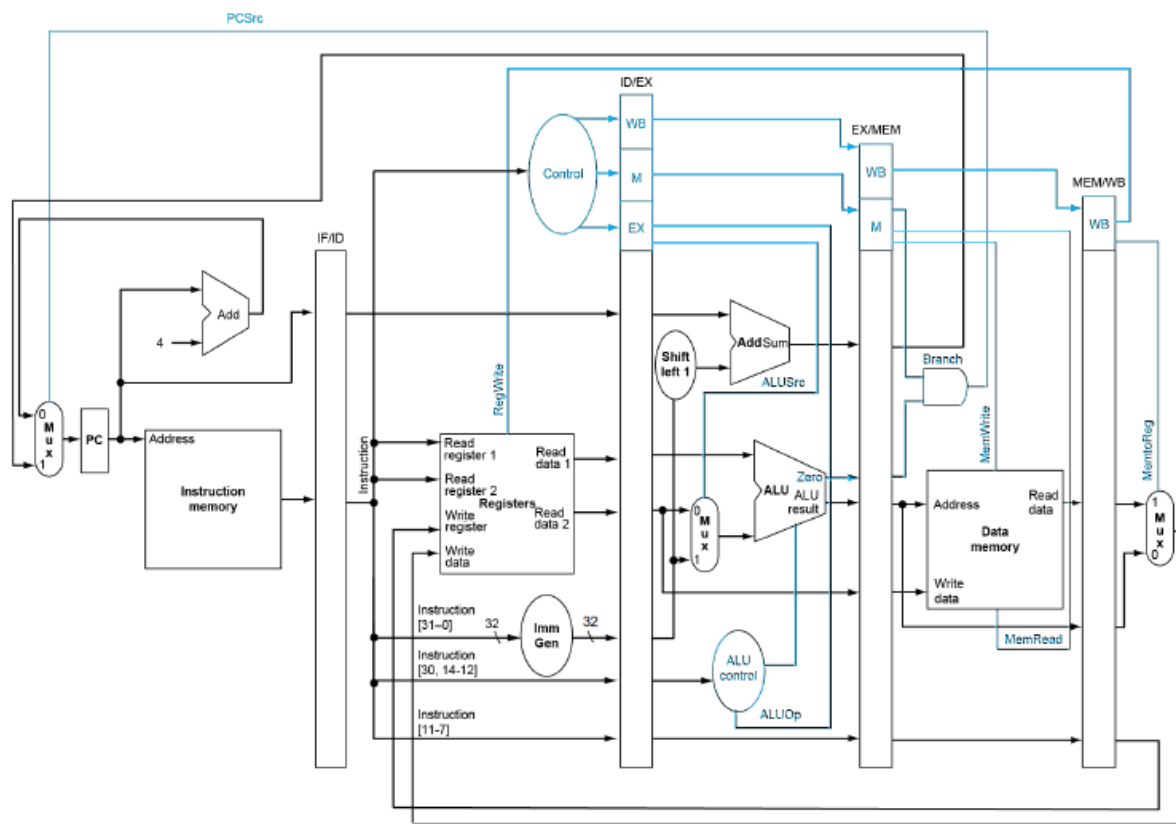
- Combinational logic does the computation during clock cycles (**between clock edges**).
- Among all kinds of computations, **longest delay determines clock period**

Pipelined Processor

- **Divide** the combinational logic into smaller pieces, so that each piece is finished in shorter time.
- Five stages, **one step per stage per cycle**:
 - **IF**: Instruction fetch from memory
 - **ID**: Instruction decode & register read
 - **EX**: Execute operation or calculate address
 - **MEM**: Access memory operand
 - **WB**: Write result back to register
- Need **registers between stages** to hold information produced in previous cycle.
- Pipelining improves performance by **increasing instruction throughput**
 - Executes multiple instructions in parallel
 - Each instruction has the same latency

Pipelined Control

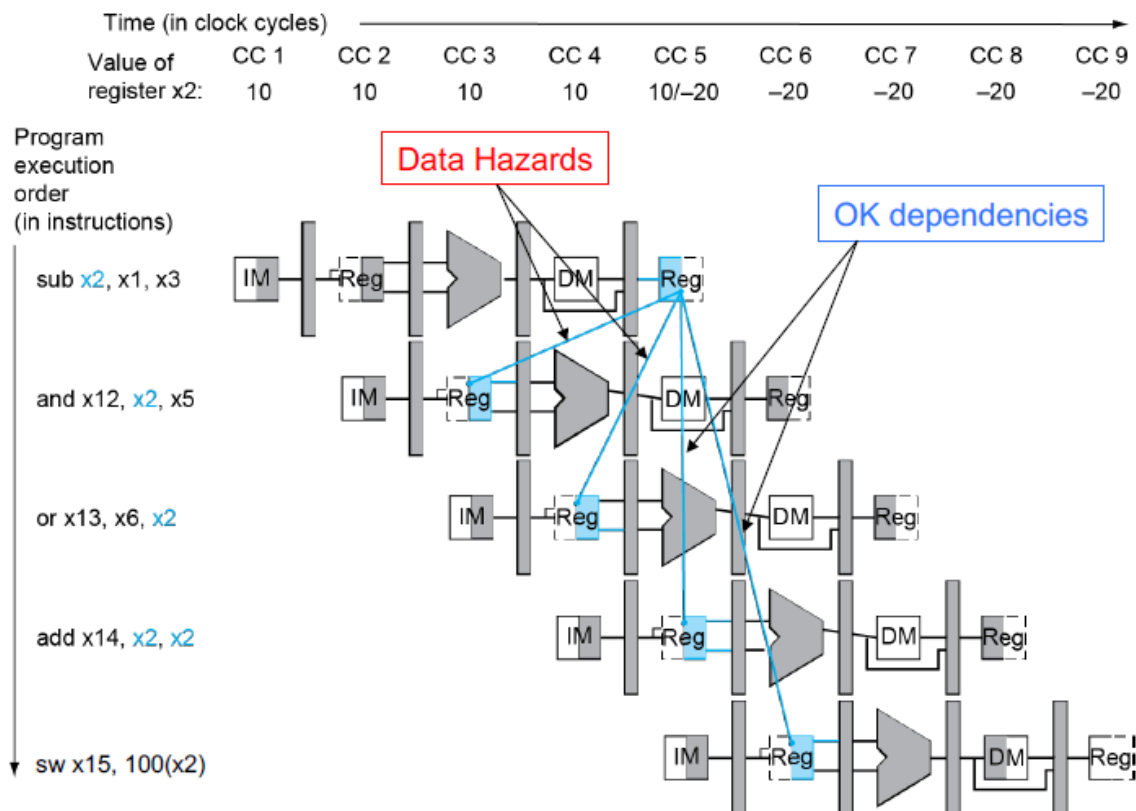
- Control signals derived from instruction
 - Passed along with corresponding instruction
 - Consumed in appropriate stages



Hazards

- Situations that prevent starting the next instruction in the next cycle
 - **Data hazard**: Need to wait for previous instruction to complete its **data read/write**
 - **Control hazard**: Decision on **control** action **depends on previous instruction**
 - **Structure hazards**: A required resource is busy

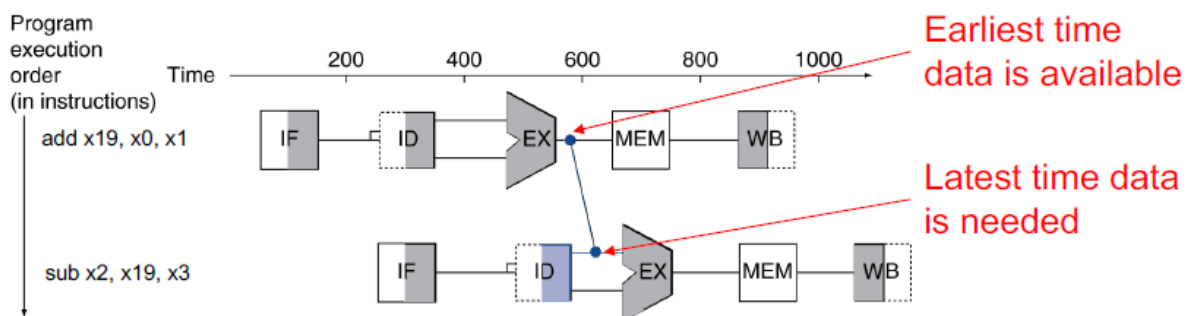
Data Hazards



- If an instruction depends on completion of a data by a previous instruction, then the instruction must be delayed by:
 - **Inserting bubbles or stalls:** Keep adding bubbles until the hazard is resolved
 - **Forwarding/Bypassing**

Forwarding

- Use result **as soon as it is available** instead of waiting for it to be stored in a register.
 - Requires extra hardware in the datapath
 - Should always **connect data from a pipeline register** instead connecting input and output directly.
 - Add mux to choose data

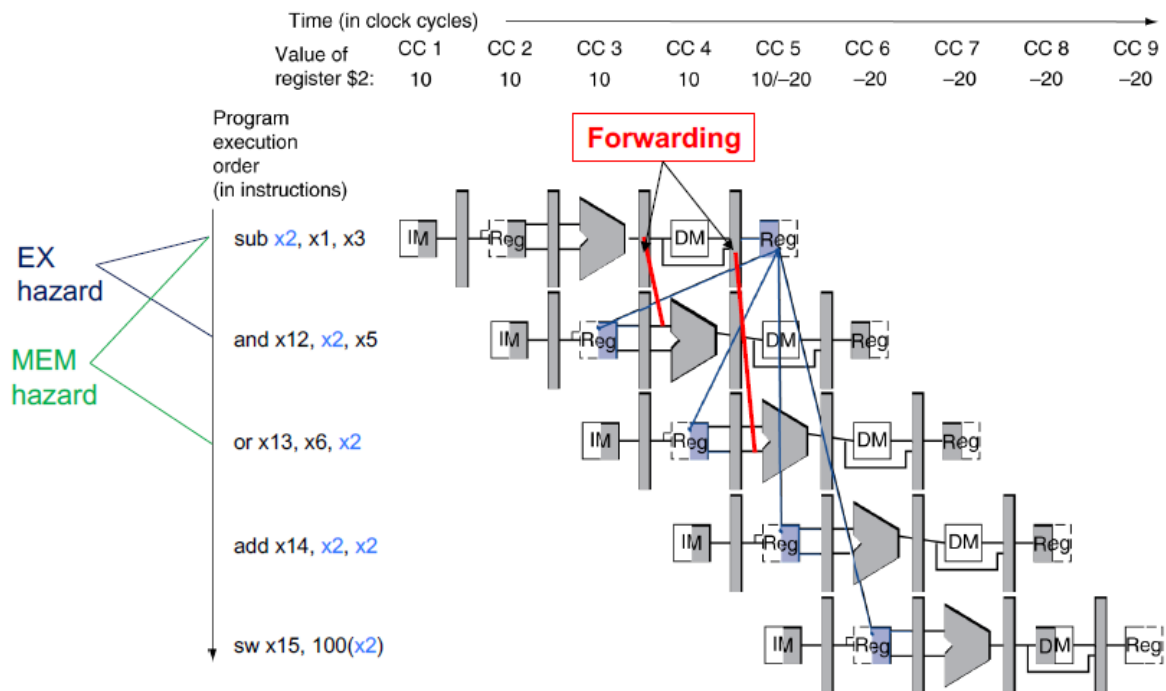


EX hazard

- Data hazard between two **adjacent** instructions
- Forwarding path from **EX/MEM** pipeline register to **ALU**

MEM hazard

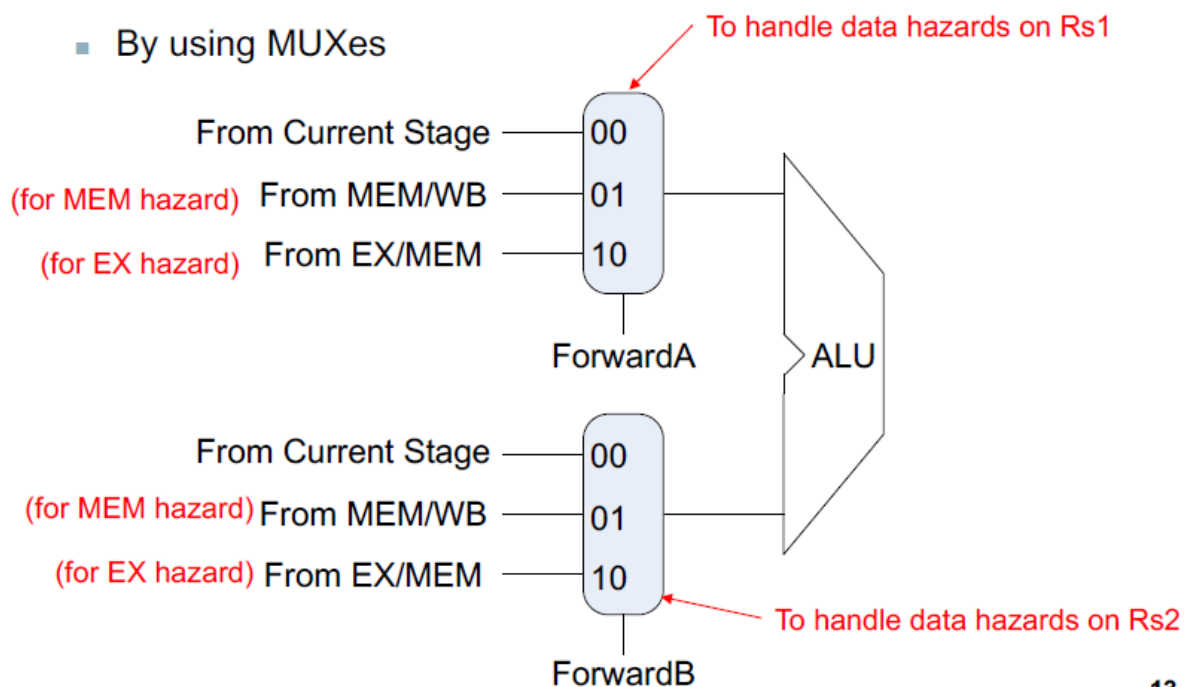
- Data hazard between two instructions with **one more instruction in between**
- Forwarding path from **MEM/WB** pipeline register to **ALU**



Forwarding Paths

Forwarding paths are created between stage pipeline registers and ALU inputs

- By using MUXes



Detecting the Need to Forward

- ALU operand register numbers in EX stage are given by: ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when:
 - EX:
 - EX/MEM.RegisterRd = ID/EX.RegisterRs1
 - EX/MEM.RegisterRd = ID/EX.RegisterRs2
 - MEM:

- MEM/WB.RegisterRd = ID/EX.RegisterRs1
 - MEM/WB.RegisterRd = ID/EX.RegisterRs2
 - And: forwarding instruction will write to a register, i.e.:
 - EX/MEM.RegWrite (EX hazard) or MEM/WB.RegWrite (MEM hazard)
 - And: Rd for that instruction is not x0, i.e.:
 - EX/MEM.RegisterRd \neq 0 (EX hazard) or MEM/WB.RegisterRd \neq 0 (MEM hazard)
- Design Forwarding Unit to judge the condition and output select signals (ForwardA, ForwardB) for mux.

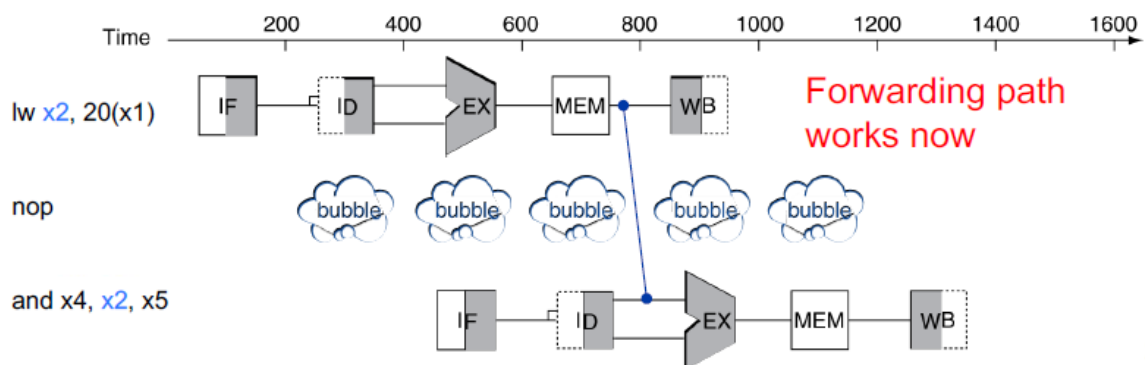
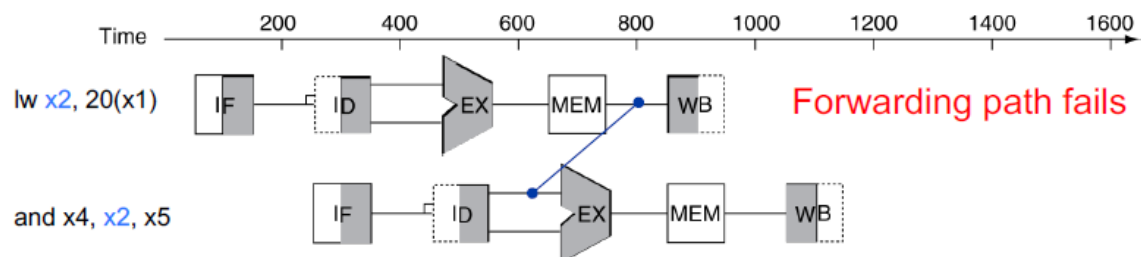
Double Data Hazard

- Conditions for both hazards are satisfied
 - e.g.:


```
add x1 x1 x2
add x1 x1 x3
add x1 x1 x4
```
- Revise MEM hazard condition that **only forward if EX hazard condition isn't true**

Load-Use Hazard Detection

- load instruction: needs stalls as well as forwarding
 - Indicated by: ID/EX.MemRead
- Load-use hazard when:
 - ID/EX.MemRead and ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs2))
- Hazard Detection unit** is added in ID stage. If detected, **stall and insert bubble**.



- To stall the pipeline:
 - Force control signals in ID/EX pipeline register to 0's
 - Prevent updates of PC and IF/ID registers

