Central Processing Unit and Computer Arithmetic

Computer Architecture — Module 4 Notes

1. Instruction Formats

1.1 Types of Instruction Formats

An instruction format defines the layout of bits in an instruction word.

Elements of an instruction:

- Operation Code (Opcode): specifies the operation to perform
- Address Field: memory address or register number
- Mode Field: specifies the addressing mode

Format Type	Description	Advantages	Disadvantages
Zero-Address	Stack-based operations	Compact code	Limited operations
One-Address	Single operand (Accumulator)	Simple CPU	More instructions needed
Two-Address	Two operands	Flexible	Longer instructions
Three-Address	Three operands	More powerful	Long instructions

1.2 Zero-Address Instructions (Stack Architecture)

```
Format: [Opcode]

Example:
PUSH X    ; Push X onto stack
PUSH Y    ; Push Y onto stack
ADD       ; Add top two stack elements
POP Z     ; Pop result to Z
```

Operation:

```
Stack before: [... X Y]
ADD: Y + X → Result
Stack after: [... Result]
```

1.3 One-Address Instructions (Accumulator Architecture)

```
Format: [Opcode] [Address]
```

```
Example:
LOAD X    ; AC ← X
ADD Y     ; AC ← AC + Y
STORE Z   ; Z ← AC
```

Operation:

```
AC = Accumulator
LOAD X: AC = M[X]
ADD Y: AC = AC + M[Y]
STORE Z: M[Z] = AC
```

1.4 Two-Address Instructions

```
Format: [Opcode] [Destination] [Source]

Example:
ADD R1, R2   ; R1 ← R1 + R2
SUB R3, R4   ; R3 ← R3 - R4
```

Operation:

```
Destination ← Destination Operation Source
```

1.5 Three-Address Instructions

```
Format: [Opcode] [Destination] [Source1] [Source2]

Example:
ADD R1, R2, R3   ; R1 ← R2 + R3
MUL R4, R5, R6   ; R4 ← R5 × R6
```

Operation:

```
Destination ← Source1 Operation Source2
```

1.6 Instruction Format Comparison

Expression	Zero-Addr	One-Addr	Two-Addr	Three-Addr
X = A + B	PUSH A / PUSH B / ADD / POP X	LOAD A / ADD B / STORE X	ADD X, A, B	ADD X, A, B
X = (A+B)×C	PUSH A / PUSH B / ADD /	LOAD A / ADD B / MUL	MUL X, A, B /	MUL X, A, B /

Expression	Zero-Addr	One-Addr	Two-Addr	Three-Addr
	PUSH C / MUL / POP X	C / STORE X	ADD X, C	ADD X, C

2. Addressing Modes

Addressing modes specify how to locate the operand.

2.1 Types of Addressing Modes

Mode	Format	Description	Example
Immediate	#Value	Operand is in instruction	ADD #5
Direct	Address	Operand at memory address	ADD 100
Indirect	@Address	Address contains operand address	ADD @100
Register	Register	Operand is in register	ADD R1
Register Indirect	(Register)	Register holds operand address	ADD (R1)
Indexed	Address(Register)	Address + Register offset	ADD 100(R1)
Relative	Address	PC-relative addressing	ADD 100
Base Register	Address(Register)	Base + Displacement	ADD 100(R1)

2.2 Detailed Explanation

Immediate Addressing

```
Opcode   |   Operand = 5
ADD      |   5
↓        |
AC = AC + 5
```

- Fast execution
- Limited operand size (part of instruction)

Direct Addressing

```
Opcode   |   Address = 200
ADD      |   200
↓        |
AC = AC + M[200]
```

- Simple addressing
- Limited address space

Indirect Addressing

```
Opcode    |    Address = 200
ADD       |    @200
↓         |
Address = M[200] = 300
AC = AC + M[300]
```

- Large address space
- Multiple memory access (slower)

Register Addressing

```
Opcode    |    Register = 3
ADD       |    R3
↓         |
AC = AC + R3
```

- Fast execution
- Limited number of registers

Register Indirect Addressing

```
Opcode    |    Register = 3
ADD       |    (R3)
↓         |
Address = R3 = 200
AC = AC + M[200]
```

- Large address space
- Flexible pointer manipulation

Indexed Addressing

```
Opcode    |    Address = 100, Index = R1
ADD       |    100(R1)
↓         |
Effective Address = 100 + R1
AC = AC + M[EA]
```

- Useful for arrays
- Base + offset calculation

Relative Addressing

```
Opcode    |    Address = 50
ADD       |    50
↓         |
EA = PC + 50
AC = AC + M[EA]
```

- Position-independent code

- Used in branch instructions

Base Register Addressing

```
Opcode    |    Address = 100, Base = R1
ADD       |    100(R1)
↓         |
EA = 100 + R1
AC = AC + M[EA]
```

- Segment management
- Memory protection

2.3 Addressing Mode Comparison

Mode	Speed	Flexibility	Address Range	Example
Immediate	Fastest	Low	Limited	Constants
Direct	Fast	Low	Limited	Simple variables
Indirect	Slow	High	Large	Pointers
Register	Fastest	Low	N/A	Variables in registers
Register Indirect	Fast	High	Large	Arrays, pointers
Indexed	Medium	High	Large	Array access
Relative	Fast	Medium	Limited	Branch instructions

3. Stack Operations

3.1 Stack Organization

Stack: A Last-In-First-Out (LIFO) data structure.

```
Memory:
+---------------+
|    ↓          |
|  SP →         |  ← Top of stack
|  Data 3       |
|  Data 2       |
|  Data 1       |
+---------------+
```

Stack Operations:

```
PUSH: SP ← SP - 1; M[SP] ← Data
POP:  Data ← M[SP]; SP ← SP + 1
```

3.2 Push Operation

Algorithm:

```
1. SP ← SP - 1
2. M[SP] ← Data
3. Update flags
```

Example:

```
Initial SP = 2000
PUSH 100:

Step 1: SP = 1999
Step 2: M[1999] = 100
Result: SP = 1999
```

3.3 Pop Operation

Algorithm:

```
1. Data ← M[SP]
2. SP ← SP + 1
3. Update flags
```

Example:

```
Initial SP = 1999
POP R1:

Step 1: R1 = M[1999] = 100
Step 2: SP = 2000
Result: SP = 2000, R1 = 100
```

3.4 Stack Applications

Expression Evaluation

```
Infix: A + B × C
Postfix: A B C × +
Using Stack:
1. Push A
2. Push B
3. Push C
4. Mul → Pop B, C, Push B×C
5. Add → Pop A, B×C, Push A+B×C
```

Function Calls

- Save return address
- Save frame pointer
- Allocate local variables

Interrupt Handling

- Save program counter
- Save processor state
- Handle interrupt
- Restore state

4. Data Transfer and Manipulation

4.1 Data Transfer Instructions

Instruction	Example	Description
MOV	MOV R1, R2	Copy R2 to R1
LOAD	LOAD R1, [100]	Load from memory to register
STORE	STORE [100], R1	Store register to memory
LDI	LDI R1, #5	Load immediate value
PUSH	PUSH R1	Push register to stack
POP	POP R1	Pop stack to register
XCHG	XCHG R1, R2	Exchange registers
MOVS	MOVS R1, R2	Move string

4.2 Data Manipulation Instructions

Arithmetic Instructions

```
ADD R1, R2     ; R1 ← R1 + R2
SUB R1, R2     ; R1 ← R1 - R2
MUL R1, R2     ; R1 ← R1 × R2
DIV R1, R2     ; R1 ← R1 ÷ R2
INC R1         ; R1 ← R1 + 1
DEC R1         ; R1 ← R1 - 1
```

Logical Instructions

```
AND R1, R2     ; R1 ← R1 & R2
```

```
OR R1, R2      ; R1 ← R1 | R2
XOR R1, R2     ; R1 ← R1 ⊕ R2
NOT R1         ; R1 ← ~R1
```

Shift Instructions

```
SHL R1, #3     ; R1 ← R1 << 3
SHR R1, #2     ; R1 ← R1 >> 2
ROL R1, #1     ; R1 ← R1 rotated left 1
ROR R1, #1     ; R1 ← R1 rotated right 1
```

4.3 Comparison Instructions

Instruction	Description	Flags Set
CMP	Compare two operands	Z, N, C, V
TEST	Logical AND (no result)	Z, N, C, V

```
CMP R1, R2     ; Compare R1 with R2
; Sets flags:
; Z = 1 if R1 = R2
; N = 1 if R1 < R2
; C = 1 if R1 < R2 (unsigned)
```

5. Program Control

5.1 Branch Instructions

Unconditional Branch:

```
JMP Address    ; Jump to Address
BRA Label      ; Branch to Label (relative)
JMP R1         ; Jump to address in R1
```

Conditional Branch:

```
BNE Label      ; Branch if Not Equal
BEQ Label      ; Branch if Equal
BLT Label      ; Branch if Less Than
BGT Label      ; Branch if Greater Than
BLE Label      ; Branch if Less or Equal
BGE Label      ; Branch if Greater or Equal
```

5.2 Branch Conditions

Condition	Flags	Signed	Unsigned
Equal	Z=1	Equal	Equal
Not Equal	Z=0	Not Equal	Not Equal
Greater Than	Z=0, N=V	>	>
Less Than	N≠V	<	<
Greater or Equal	N=V	≥	≥
Less or Equal	Z=1 or N≠V	≤	≤

5.3 Loop Control

Counted Loop:

```
    MOV R1, #10    ; Initialize counter
Loop:  ...
    DEC R1         ; Decrement counter
    BNE Loop       ; Branch if not zero
```

Conditional Loop:

```
Loop:  CMP R1, #100   ; Check condition
    BGT Exit       ; Branch if greater
    ...
    INC R1         ; Increment
    JMP Loop       ; Repeat
Exit:  ...
```

5.4 Subroutine Instructions

Instruction	Description
CALL	Branch to subroutine
RET	Return from subroutine
CALLR	Relative subroutine call
TRAP	Software interrupt

Subroutine Example:

```
Main:  ...
    CALL Subroutine
    ...
    HALT
```

```
Subroutine:
    PUSH R1      ; Save registers
    ...
    POP R1       ; Restore registers
    RET
```

5.5 Interrupt Control

```
INT Address    ; Software interrupt
IRET           ; Return from interrupt
CLI            ; Clear interrupt flag
STI            ; Set interrupt flag
```

6. RISC and CISC Processors

6.1 CISC (Complex Instruction Set Computer)

Characteristics:

- Large instruction set
- Variable instruction length
- Complex instructions
- Microprogrammed control
- High memory usage

Examples: x86, IBM 370, VAX

Advantages:

- Smaller program size
- Fewer memory references
- Complex operations in hardware

Disadvantages:

- Complex hardware
- Slower execution
- Less efficient pipelining

6.2 RISC (Reduced Instruction Set Computer)

Characteristics:

- Small instruction set
- Fixed instruction length
- Simple instructions

- Hardwired control
- Load/store architecture

Examples: ARM, MIPS, RISC-V

Advantages:

- Fast execution
- Simple hardware
- Efficient pipelining
- Lower power consumption

Disadvantages:

- Larger program size
- More memory references
- Compiler dependent

6.3 RISC vs CISC Comparison

Feature	RISC	CISC
Instruction Set	Small, simple	Large, complex
Instruction Length	Fixed	Variable
Clock Cycles	1 per instruction	Multiple per instruction
Control Unit	Hardwired	Microprogrammed
Memory Access	Load/Store architecture	Memory-to-memory allowed
Addressing Modes	Few	Many
Register Set	Large	Small
Pipelining	Efficient	Limited
Examples	ARM, MIPS	x86, VAX

6.4 RISC Pipeline Example

```
5-Stage RISC Pipeline:
+-----+-----+-----+-----+-----+
| IF  | ID  | EX  | MEM | WB  |
+-----+-----+-----+-----+-----+

IF: Instruction Fetch
ID: Instruction Decode
EX: Execute
MEM: Memory Access
WB: Write Back
```

Pipeline Hazards:

- Structural: Resource conflict
- Data: Data dependency
- Control: Branch instructions

7. Pipelining

7.1 Pipeline Basics

Pipelining: Overlapping instruction execution to increase throughput.

Ideal Speedup:

```
Speedup = (Number of Pipeline Stages) / (1 + Stall Cycles)
```

Example: 5-stage pipeline, Speedup ≈ 5x (in ideal conditions)

7.2 Pipeline Hazards

1. Structural Hazard

- Resource conflicts
- Solution: Duplicate resources

```
Instruction 1: IF ID EX MEM WB
Instruction 2:    IF ID EX MEM WB
                  ↑
            Conflict (same memory)
```

2. Data Hazard

```
ADD R1, R2, R3   ; R1 = R2 + R3
SUB R4, R1, R5   ; R4 = R1 - R5 (depends on R1)
```

Forwarding (Bypassing):

```
ADD R1, R2, R3   ; IF ID EX MEM WB
SUB R4, R1, R5   ;    IF ID EX MEM WB
                   ↑
                Forwarding
```

3. Control Hazard

```
Branch instruction
```

```
- Need to know target address
- Solution: Branch prediction
```

7.3 Superscalar and Superpipelining

Superscalar:

- Multiple instructions per cycle
- Multiple execution units

```
Cycle 1: I1, I2
Cycle 2: I3, I4
Cycle 3: I5, I6
```

Superpipelining:

- Deep pipelines (more stages)
- Faster clock speed

8. Computer Arithmetic Algorithms

8.1 Addition and Subtraction

Addition:

```
A + B = Result

Binary Addition:
  0110 (6)
+ 0111 (7)
  ----
  1101 (13)

For signed numbers using 2's complement:
  0011 (3)
+ 1110 (-2)
  ----
  0001 (1)
```

Subtraction (using 2's complement):

```
A - B = A + (-B)

Step 1: Find 2's complement of B
Step 2: Add to A
Step 3: Discard final carry
```

```
Example: 5 - 3
0101 (5)
+ 1101 (2's complement of 3)
 ----
10010 → 0010 (2)
```

8.2 Multiplication Algorithms

Paper-and-Pencil Method:

```
  1101 (13)
× 1011 (11)
--------
  1101
 1101
0000
1101
--------
10001111 (143)
```

Booth's Multiplication Algorithm

Purpose: Efficient multiplication of signed numbers.

Algorithm:

- Initialize: Q = Multiplier, M = Multiplicand, A = 0, $Q_{-1} = 0$, Count = n
- For each bit: check Q_0 and Q_{-1}
- If $Q_0Q_{-1} = 01$: A = A + M
- If $Q_0Q_{-1} = 10$: A = A − M
- If $Q_0Q_{-1} = 00$ or 11: No operation
- Arithmetic shift right A, Q, Q_{-1}; decrement Count
- Result = AQ

Example: 5 × (−3) using Booth's Algorithm

```
Initial:
M = 0011 (3)
-Q = 1101 (-3)
A = 0000
Q = 1101 (-3)
Q-1 = 0
Count = 4

Step 1: Q0Q-1 = 10 → A = A - M = 0000 - 0011 = 1101
      ASR → A = 1110, Q = 1110, Q-1 = 1

Step 2: Q0Q-1 = 01 → A = A + M = 1110 + 0011 = 0001
      ASR → A = 0000, Q = 1111, Q-1 = 0
```

```
Step 3: Q0Q-1 = 10 → A = A - M = 0000 - 0011 = 1101
      ASR → A = 1110, Q = 1111, Q-1 = 1

Step 4: Q0Q-1 = 11 → No operation
      ASR → A = 1111, Q = 0111, Q-1 = 1

Result = AQ = 11110111 = -15 × 5
```

8.3 Division Algorithms

Restoring Division

Algorithm:

- Initialize: A = Dividend (n bits), M = Divisor (n bits), Q = 0, Count = n
- For each bit: left shift AQ; A = A − M
- If A ≥ 0: $Q_0 = 1$
- Else: A = A + M (Restore); $Q_0 = 0$
- Decrement Count
- Quotient = Q, Remainder = A

Non-Restoring Division

Algorithm:

- Initialize: A = Dividend, M = Divisor, Count = n
- If A ≥ 0: left shift AQ; A = A − M; $Q_0 = 1$
- Else: left shift AQ; A = A + M; $Q_0 = 0$
- If A < 0 at the end: A = A + M

Example: 9 ÷ 3 using Restoring Division

```
9 = 1001, 3 = 0011
A = 0000, Q = 1001, M = 0011

Step 1: Shift AQ → A = 0001, Q = 0010
      A - M = 0001 - 0011 = -0010 (Negative)
      Restore: A = 0001, Q0 = 0

Step 2: Shift AQ → A = 0010, Q = 0100
      A - M = 0010 - 0011 = -0001 (Negative)
      Restore: A = 0010, Q0 = 0

Step 3: Shift AQ → A = 0100, Q = 1000
      A - M = 0100 - 0011 = 0001 (Positive)
      Q0 = 1

Step 4: Shift AQ → A = 0011, Q = 0001
```

A - M = 0011 - 0011 = 0000 (Positive)
Q0 = 1

Result: Quotient = 0011 (3), Remainder = 0000 (0)

8.4 Floating-Point Arithmetic

Floating-Point Addition

1. Align exponents
2. Add mantissas
3. Normalize result
4. Round result

Example: 25.625 + 12.5
$25.625 = 1.1001101 \times 2^4$
$12.5 = 1.1001000 \times 2^3$

Step 1: Align exponents (4 and 3)
$1.1001000 \times 2^3 = 0.1100100 \times 2^4$

Step 2: Add mantissas
1.1001101 + 0.1100100 = 10.0110001

Step 3: Normalize
$10.0110001 \times 2^4 = 1.00110001 \times 2^5$

Step 4: Round
25.625 + 12.5 = 38.125

Floating-Point Multiplication

1. Multiply mantissas
2. Add exponents
3. Normalize result
4. Round result

Example: 25.625 × 12.5
$25.625 = 1.1001101 \times 2^4$
$12.5 = 1.1001000 \times 2^3$

Step 1: Multiply mantissas
1.1001101 × 1.1001000 = 10.000111...

Step 2: Add exponents
$2^4 \times 2^3 = 2^7$

Step 3: Normalize
$10.000111... \times 2^7 = 1.0000111... \times 2^8$

Step 4: Round
25.625 × 12.5 = 320.3125

9. Important Formulas and Equations

9.1 Pipeline Speedup

Speedup = n / (1 + (n-1)×StallRate)
where n = number of pipeline stages

9.2 Booth's Multiplication Algorithm

Booth's Algorithm:
For multiplier bits:
if Q_0Q_{-1} = 00 or 11: No operation
if Q_0Q_{-1} = 01: A = A + M
if Q_0Q_{-1} = 10: A = A - M
Then ASR AQ

9.3 Division Algorithm

Restoring Division:
1. Shift AQ left
2. A = A - M
3. If A < 0:
   A = A + M
   Q_0 = 0
   else Q_0 = 1

9.4 Floating-Point Operations

Addition: (M1 × 2^E1) + (M2 × 2^E2)
= (M1 + M2 × 2^(E2-E1)) × 2^E1
(after aligning exponents)

Multiplication: (M1 × 2^E1) × (M2 × 2^E2)
= (M1 × M2) × 2^(E1+E2)

10. Practice Problems

Problem 1: Instruction Format

Design instruction format for:

- 16-bit instruction
- 4-bit opcode
- 12-bit address

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+--+--+--+--+----------------------------+
|     Opcode     |      Address           |
|     (4-bit)    |      (12-bit)          |
+----------------+------------------------+
```

Problem 2: Addressing Mode

Find effective address for ADD [200], R1 where: R1 = 100, M[200] = 300

```
EA = M[200] + R1
   = 300 + 100
   = 400
Operand = M[400]
```

Problem 3: Pipeline

Calculate speedup for 5-stage pipeline with 20% branch stalls

```
Speedup = n / (1 + (n-1)×StallRate)
        = 5 / (1 + 4×0.2)
        = 5 / (1 + 0.8)
        = 5 / 1.8
        = 2.78x
```

Problem 4: Booth's Algorithm

Multiply 7 × (−5) using Booth's algorithm

```
M = 0111 (7)
-Q = 1011 (-5)
A = 0000, Q = 1011, Q-1 = 0

Step 1: Q0Q-1 = 10 → A = A - M = 0000 - 0111 = 1001
        ASR → A = 1100, Q = 1101, Q-1 = 1

Step 2: Q0Q-1 = 11 → No operation
        ASR → A = 1110, Q = 0110, Q-1 = 1
```

Step 3: Q0Q-1 = 01 → A = A + M = 1110 + 0111 = 0101
ASR → A = 0010, Q = 1011, Q-1 = 0

Step 4: Q0Q-1 = 10 → A = A - M = 0010 - 0111 = 1011
ASR → A = 1101, Q = 1101, Q-1 = 1

Result: AQ = 11011101 = -35 (7 × -5 = -35)

11. Key Takeaways

✔️ Instruction formats vary from zero-address to three-address
✔️ Addressing modes determine how operands are accessed
✔️ Stack operations provide efficient expression evaluation
✔️ Control flow instructions handle program branching and loops
✔️ RISC processors use simple instructions, fixed format
✔️ CISC processors use complex instructions, variable format
✔️ Pipelining increases throughput but introduces hazards
✔️ Booth's algorithm efficiently multiplies signed numbers
✔️ Floating-point arithmetic requires exponent alignment
✔️ Computer arithmetic handles addition, multiplication, division

These comprehensive notes cover Module 4 of Computer Architecture. Focus on understanding instruction formats, addressing modes, pipeline concepts, and arithmetic algorithms. Practice Booth's algorithm and floating-point operations for better understanding.