



# **Chapter 8**

## **INSTRUCTION SET ARCHITECTURE**

# Chapter 8: Instruction Set architecture

- Introduction
- Type of Instructions
- High level language program to execution
- Instruction cycle
- Instruction Set architecture (ISA)
- Addressing Modes and Machine code
- Types of ISA
- Design Examples ACC ISA
- Sparc and Pentium Assembly program architecture
- Performance Parameters

# Chapter 8: Introduction

- The preceding chapters covered digital design concepts and , Datapath and Memory organization
- Modern CPUs implement pipelining and instruction-level parallelism (ILP) to increase performance
- Data path fetches an instruction from memory and decodes the instruction

# Chapter 8: Introduction (cont'd)

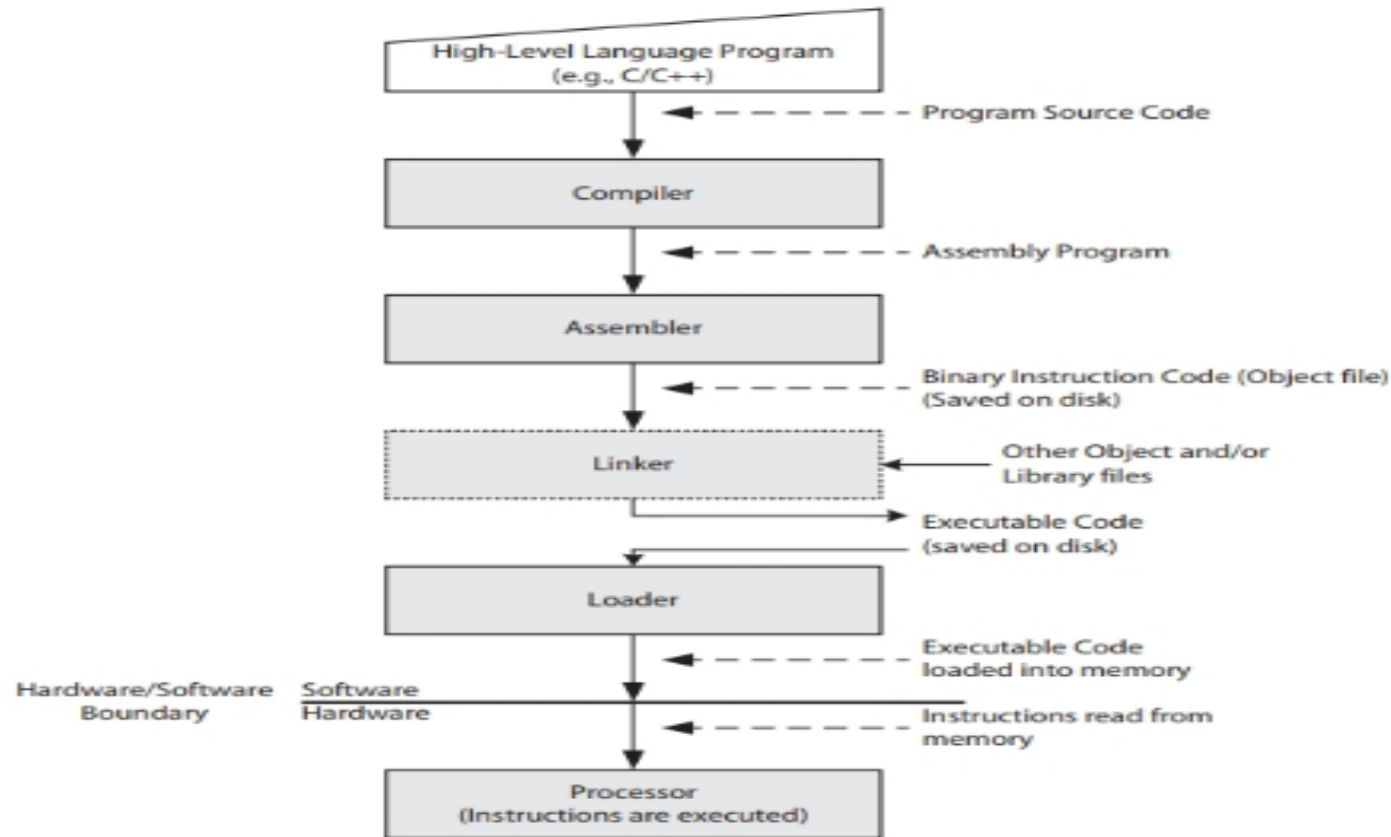
- An instruction set architecture (ISA) refers to Data path that executes a program.
  - Single cycle
  - Multicycle
  - Pipeline
- Data-dependent instructions go through the pipeline stages
  - Can lead to stall in the pipeline
  - May reduce pipeline efficiency
- Branch instructions change execution flow
  - jne (Jump if not equal to)
  - Jge (Jump if greater than or equal to)
  - Jmp (Will jump to specified address)

# Type of Instructions

- **A processor is designed for general-purpose programming**
  - **Graphics processing Units (GPU)**
  - **Digital signal processing (DSP)**
- **Special purpose instructions**
  - **SIMD**
  - **Computer security related instructions**
- **Data-manipulation instructions**
- **Data-movement instructions**
- **Program-flow control instructions**

# High level language program to execution

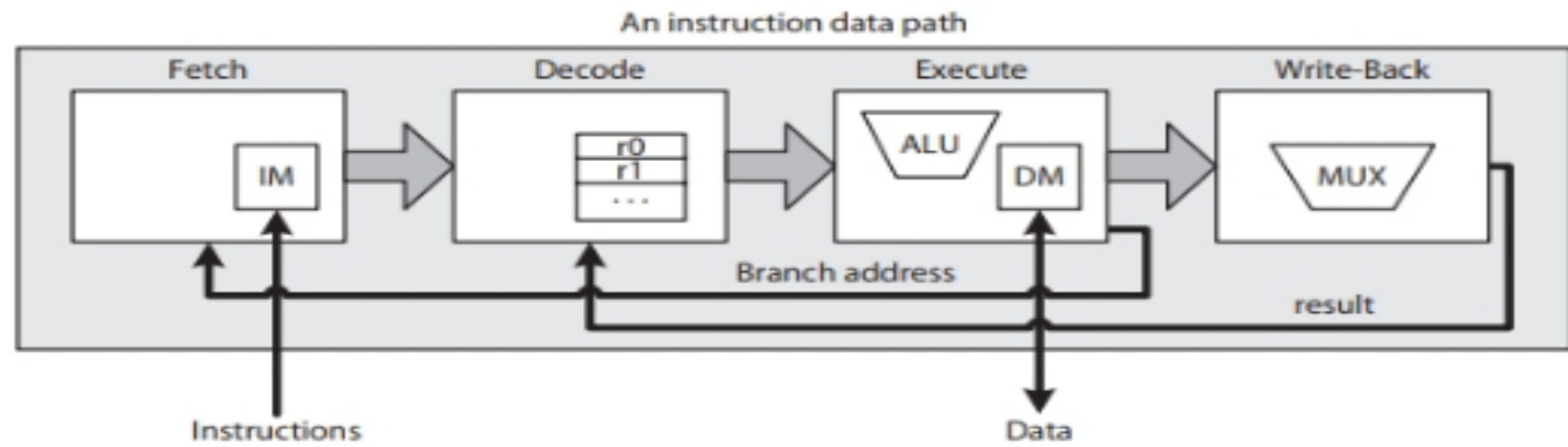
As shown in Fig. 8.1, a software program is typically written in a high-level language, such as C/C++ or Java, and is translated by a compiler into assembly



**FIGURE 8.1** Basic program translation and execution process.

# Instruction Cycle

- **Data path has four main tasks**
  - **Fetch**
  - **Decode**
  - **Execute**
    - **May access data memory (another cache)**
  - **Writes Results**



**FIGURE 8.2** An instruction data path with instruction memory (IM) and data memory (DM).

# Instruction Set architecture (ISA)

- **Types of Instruction set architecture:**
  - **Stack ISA**
  - **Accumulator ISA**
  - **CISC ISA (complex instruction set computer)**
  - **RISC ISA (reduced instruction set computer)**
- **An opcode is part of instruction set which tells the hardware what operation needs to be performed**
  - **Assembly language mnemonic form, an opcode is a command such as ADD, MOV, SUB or JMP**
  - **For example: MOV AL, 34h**
- **An instruction includes a set of operands that are specified explicitly, implicitly**



# Addressing Modes and Syntax Examples

- Immediate
  - E.g., Add R1, 9;
- Direct
  - E.g., ADD R1, (M[9]);
- Register
  - E.g., ADD R1, R2;
- Register direct
  - E.g., ADD R1, (R2);
- Register indexed
  - E.g., ADD R1, R2, (M[9]);

Operand Notation	Addressing Mode
V	I, immediate: V is an immediate input operand, a 2's complement number.
(V)	D, direct: V is a memory address and (V) indicates the content of memory address V (i.e., M[V]).
R	R, register: Indicates an input data register source or a destination register or both
R, (V)	X, indexed: V is a memory address and $R + V$ is the address of the next data item in memory (i.e., M[R + V]).

TABLE 8.1 Examples of Addressing Modes

# Machine Instruction Format (Examples)

**Stack:**

**Accumulator:**

**CISC:**

**CISC:**

**CISC:**

**CISC:**

**RISC:**

Example	Instruction
1	ADD
2	ADD 9
3	ADD R1, -9
4	ADD R1, (9)
5	ADD R1, R2, (9)
6	ADD R1, R2
7	ADD R3, R1, R2

Table 8.2

**Example**

1: ADD

2: ADD 9

3: ADD R1, -9

4: ADD R1, (9)

5: ADD R1, (R2), 9

6: ADD R1, R2

7: ADD R1, R2, R3

**Instruction Format**

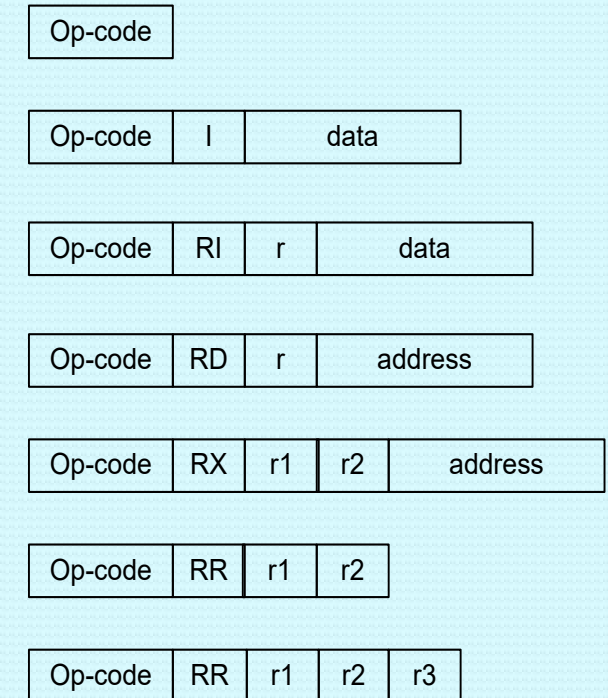


Fig 8.3

# Types of ISA – Stack (1)

- Arithmetic instructions have no explicitly declared operands
- Operands are on stack inside CPU
- E.g., ADD
- Example:  $A = B * (C + D);$ 
  - Requires converting statement into reverse polish notation
    - $CD+B*=A$
  - Reverse polish notation converted to assembly program?
- Short instructions (advantage)
- Stack as LIFO buffer (disadvantage)

# Stack ISA - Example of assembly program

- Example Program :

Instruction  
number

```
1:  PUSH (C) //stack ← M[C]
2:  PUSH (D) //stack ← M[D]
3:  ADD      //stack ← (C) + (D), values popped, added,
           //result pushed
4:  PUSH (B) //stack ← M[B]
5:  MUL      //stack ← ((C) + (D)) * (B), values popped, added,
           //result pushed
6:  POP (A)  //M[A] ← ((C) + (D)) * (B), value is popped
           //and stored in memory
```

# Types of ISA – Accumulator (2)

- One of the operands is a known register, called accumulator (Acc)
  - LD (C)
- Second operand is immediate or data from memory
- Acc always destination register
  - E.g., ADD 9 //  $\text{Acc} \leftarrow \text{Acc} + 9$
  - E.g., ADD (C) //  $\text{Acc} \leftarrow \text{Acc} + \text{M}[\text{C}]$
- Example Program:  $A = B * (C + D); ?$
- Simple data path, less hardware (advantage)
- Acc bottleneck (disadvantage)
  - E.g.,  $A = (C + D) * (E - F);$

# Acc ISA - Example of assembly program

- **Example Program:**

**1: LD       (C)**

**2: ADD     (D)**

**3: MUL     (B)**

**4: ST       (A)**

# Types of ISA – CISC (3)

- CISC (complex instruction set computer)
- Many simple and complex instructions
- Multiple addressing modes
- Many working registers (e.g., 16)
  - Recent results kept inside CPU in registers
- Arithmetic instructions can access memory
  - E.g., `ADD R1, R2 //R1 ← R1 + R2`
  - E.g., `ADD R1, (9) //R1 ← R1 + M[9]`
- Example: `A = B * (C + D);`
  - Program?
- Complex instruction set (advantage)
  - Fewer instructions per program
- Complex instruction set (disadvantage)
  - Complex data path
  - Limited pipelining of instruction cycle
  - Many instructions and addressing modes seldom used

# CISC-ISA: Example of assembly program

- Example Program:

1. LD R1, (C)
2. ADD R1, (D)
3. MUL R1, (B)
4. ST (A), R1



# Types of ISA – RISC-(4)

- RISC (reduced instruction set computer)
- Arithmetic instructions cannot access memory
- Many more working registers (e.g., 32)
- Implements only most commonly used instructions
- 3-operand instructions
  - E.g., `ADD R3, R1, R2 //R3 ← R1 + R2`
  - E.g., `ADD R2, R1, 9 //R2 ← R1 + 9`
- Only LD and ST instructions access memory
- Example Program: `A = B * (C + D);`
- Simpler and highly pipelined data path (advantage)
- Requires compiler optimization to increase efficiency
- The architecture of all modern processing cores

# RISC-ISA: Example of assembly program

- **Example Program:**

```
1. LD    R1, (C)
2. LD    R2, (D)
3. ADD   R3, R1, R2
4. LD    R4, (B)
5. MUL   R5, R3, R4
6. ST    (A), R5
```

# Design Example: Acc-ISA

- **We start with example high-level language program**
- **Design instruction set for example program**
- **Generate assembly program**
- **Generate binary machine instructions**
- **Create Acc-ISA data path**
- **Model in HDL**
- **Simulation results**

# Example Instruction format for Acc-ISA

**.code**

```
LD 0
ST (sum)
ST (i)
L1:
  CMP 7
  JGT L2
  MVX
  LD X(array)
  ADD (sum)
  ST (sum)
  LD (i)
  ADD 1
  ST (i)
  JMP L1
L2: ...
```

**.data**

```
array: RB 16
i:     RB 2
sum:   RB 2
```

## *Example code #1*

```
int array[8];
int i, sum;
sum = 0;
for (i = 0; i < 8; i++)
    sum = sum + array[i];
```

Example 8.2. The listing of an Acc-ISA assembly language program for the program in Example code 1.

# Example Program

Example code #1:

```
int array[8];
int i, sum;
sum = 0;
for (i = 0; i < 8; i++)
    sum = sum + array[i];
```

## Acc-ISA instruction set?

- Create a list of Acc-ISA instructions to translate the program to assembly language program

Op-Code	Instruction	Addressing Mode	Example	Action
0	NOP		NOP	Do nothing
1	ADD	Immediate	ADD data	$ACC \leftarrow ACC + data$
2		Direct	ADD (address)	$ACC \leftarrow ACC + M[address]$
3	CMP	Immediate	CMP data	if $ACC == data$ then $GTF = 1$ else $GTF = 0$
4	JGT	Immediate	JGT address	$PP \leftarrow address$ if $GTF = 1$
5	JMP	Immediate	JMP address	$PP \leftarrow address$
6	LD	Immediate	LD data	$ACC \leftarrow data$
7		Direct	LD (address)	$ACC \leftarrow M[address]$
8		Indexed	LD X(address)	$ACC \leftarrow M[X + address]$
9	MVX	Register	MVX	$X \leftarrow ACC$
10	ST	Direct	ST (address)	$M[address] \leftarrow ACC$

ACC: Accumulator; GTF: Greater than flag; PP: Program pointer; X: Index register

**TABLE 8.3** Example Acc-ISA Instruction Set That Translates a High-Level Program into an Equivalent Assembly Language Program

# Acc-ISA Example Assembly Program

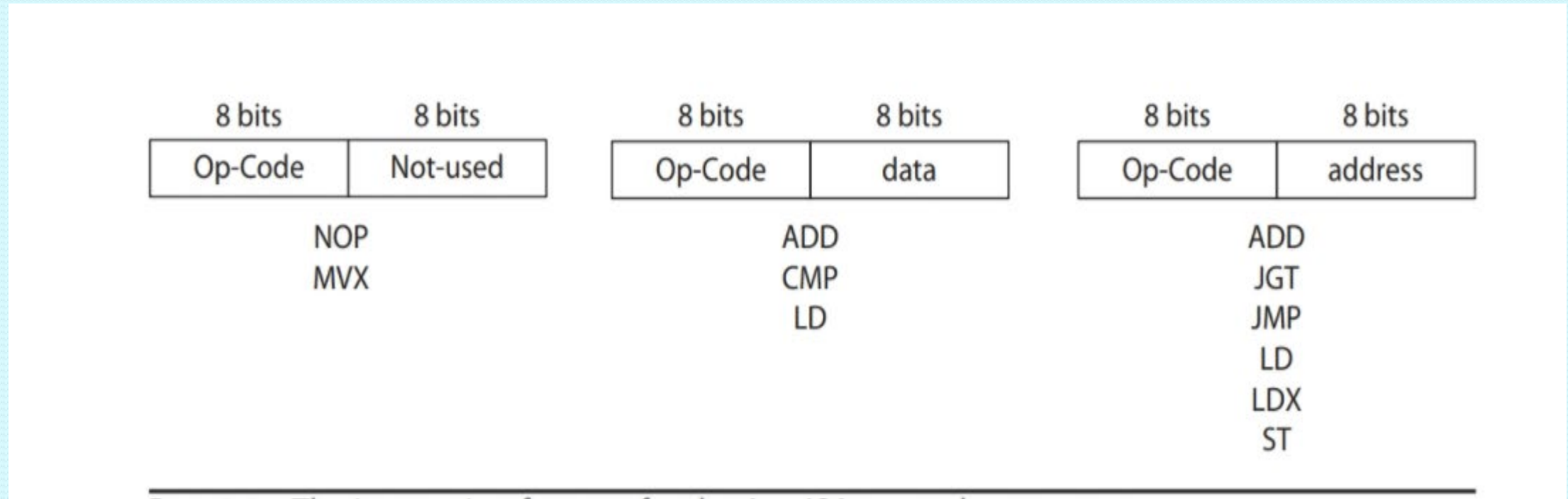


Figure 8.6 The instruction formats for the Acc-ISA example processor.

# Acc-ISA Machine Instructions

Address	Instruction		Instruction in binary	Inhex
0:	LD	0	0000,0110;0000,0000	0600
2:	ST	0xEC	0000,1010;1110,1100	0AEC
4:	ST	0xEE	0000,1010;1110,1110	0AEE
6:	CMP	7	0000,0011;0000,0111	0307
8:	JGT	0x1A	0000,0100;0001,1010	041A
A:	MVX		0000,1001;0000,0000	0900
C:	LD	X (0xF0)	0000,1000;1111,0000	08F0
E:	ADD	(0xEC)	0000,0010;1110,1100	02EC
10:	ST	(0xEC)	0000,1010;1110,1100	0AEC
12:	LD	(0xEE)	0000,0111;1110,1110	07EE
14:	ADD	1	0000,0001;0000,0001	0101
16:	ST	(0xEE)	0000,1010;1110,1110	0AEE
18:	JMP	6	0000,0101;0000,0110	0506
1A:	...			

Example 8.5. The manually assembled output for the assembly program in Example 8.2.



# Acc-ISA Pipelined Data path

- Data path consists of four stages (block diagram)

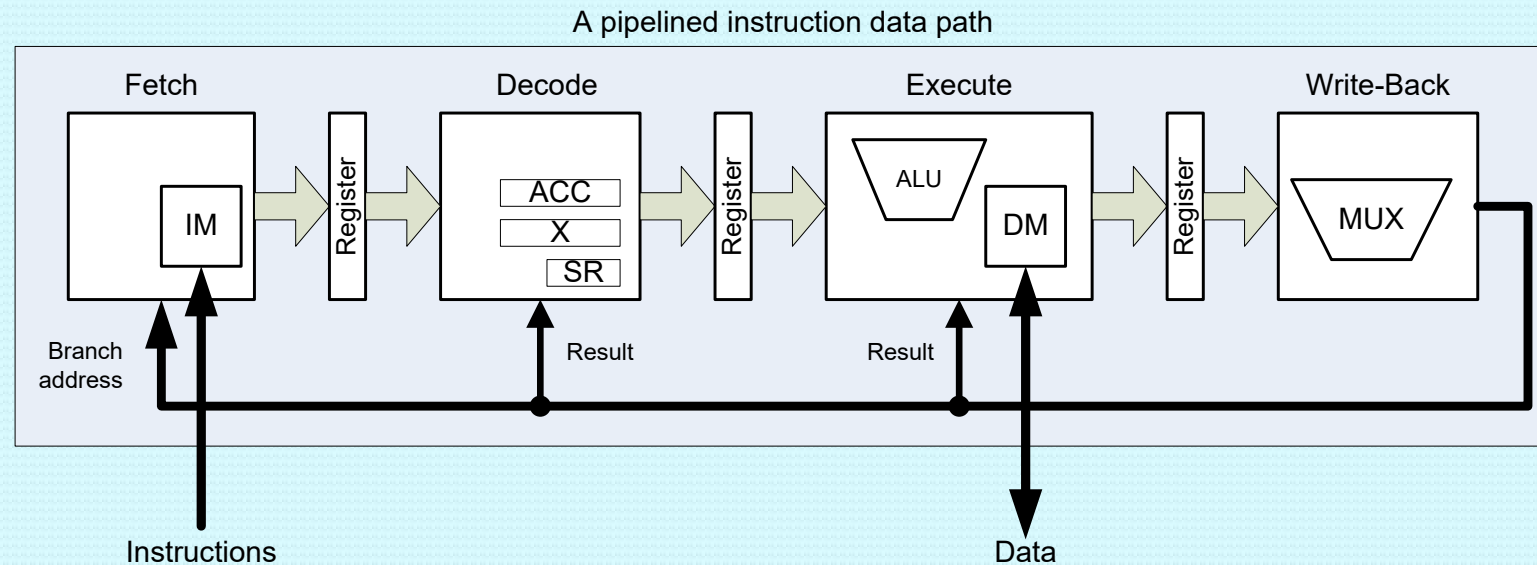
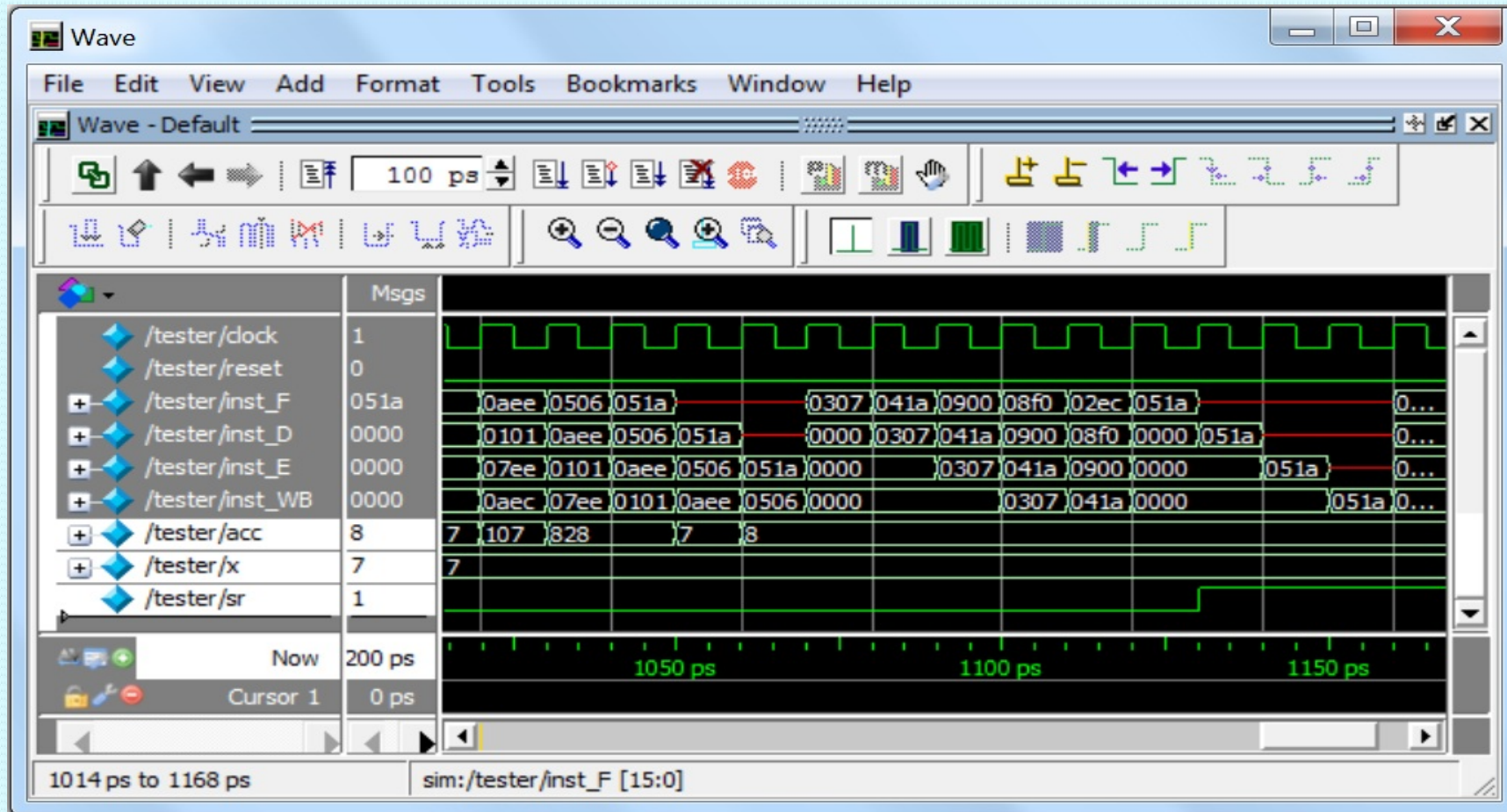


Fig 8.10



# Pipeline Simulation

- Illustrates pipeline flush on jumps



# Sparc Example Assembly Program

- Note, arithmetic instructions access only registers
- Only “ld” and “st” instructions access memory

```

        st      %g0, [%fp-48]      //Store, Memory[fp -48] ← g0 (g0 always 0) (sum = 0)
        st      %g0, [%fp-44]      //Store, Memory[fp - 44] ← g0 (i = 0)
.LL5:
        ld      [%fp-44], %g1       //Load, g1 ← Memory[i]
        cmp     %g1, 7              //Compare, is g1 > 7?
        bg     .LL6                 //Branch if greater than 7
        nop
        ld      [%fp-44], %g1       //Load, g1 ← Memory[i]
        sll     %g1, 2, %g2         //Compute ptr to array[i]: Shift Left Logical (i * 2)
        add     %fp, -8, %g1        //g1 ← fp - 8 (get memory location of array)
        add     %g2, %g1, %g1       //g1 ← g2 + g1 (array location + next i * 2)
        ld      [%fp-48], %g2       //Load sum, g2 ← Memory[fp -48]
        ld      [%g1-32], %g1       //Load array[i], g1 ← Memory[g1 - 32]
        add     %g2, %g1, %g1       //Array[i] + sum, g1 ← g2 + g1
        st      %g1, [%fp-48]       //Store sum, Memory[fp - 48] ← g1
        ld      [%fp-44], %g1       //Load i, g1 ← Memory[fp - 44]
        add     %g1, 1, %g1         //Increment, g1 ← g1 + 1
        st      %g1, [%fp-44]       //Store i, Memory[fp - 44] ← g1
        b       .LL5               //Branch to instruction ay LL5
        nop
.LL6:
```

# Pentium IV Machine instructions (CISC)

- Variable size instructions
- Requires more complex data path and control unit

```
401340:      c7 45 d0 00 00 00 00    movl    $0x0,0xffffffffd0(%ebp)
401347:      c7 45 d4 00 00 00 00    movl    $0x0,0xffffffffd4(%ebp)
40134e:      83 7d d4 07              cmpl    $0x7,0xffffffffd4(%ebp)
401352:      7f 13                    jg      401367 <_main+0x77>
401354:      8b 45 d4                mov     0xffffffffd4(%ebp),%eax
401357:      8b 54 85 d8            mov     0xffffffffd8(%ebp,%eax,4),%edx
40135b:      8d 45 d0                lea     0xffffffffd0(%ebp),%eax
40135e:      01 10                    add     %edx,(%eax)
401360:      8d 45 d4                lea     0xffffffffd4(%ebp),%eax
401363:      ff 00                    incl    (%eax)
401365:      eb e7                  jmp     40134e <_main+0x5e>
401367:      b8 02 00 00 00          ...
```

# Performance Parameters

- **Cycles per instruction**
  - **Average number of clock cycles used to execute each instruction**

$$CPI = \frac{\text{Number of clock cycles used } (N)}{\text{Number of instructions executed } (n)}$$

- **Execution time (T) of a program**

$$T = CPI * n * \tau$$

# Supplemental Slides

# Acc-ISA Pipelined Data Path

