# Chapter 8
## INSTRUCTION SET ARCHITECTURE



## In this Chapter

- Instruction set architecture (ISA) defined
- Instruction execution cycle
- Types of ISA
- CPU data path design examples
- Performance parameters
- Performance improvement techniques



## Instruction Set Architecture (ISA)

- Defines CPU data path in terms of RTNs for set of instructions
- Defines different types of instructions and addressing modes as instruction set
  - Data movement instructions
  - Data manipulation instructions
  - Program-flow instructions
- Instructions sufficient to translate high-level language programs to assembly language programs



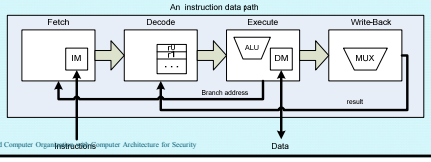
## From high level language program to execution

- Compiler: Translates program to assembly instructions
- Assembler: Translates instructions to machine instructions (binary)
- Linker: Links with external routines to create executable program
- Loader: Loads executable codes to memory for execution
- CPU: Fetches instructions and data from memory and executes instructions
  - Known as Instruction Cycle



## Instruction Cycle (Data path has four main tasks)

- Fetch: Gets next sequential instruction from instruction memory (a cache)
- Decode: Figures out what control signals should be generated
- Execute: Data path performs RTNs required by instruction
  - May access data memory (another cache)
- Writes Results: Stores obtained result (typically one, if any) in register






## Addressing Modes and Syntax Examples

- Immediate
  - Operand immediately available from instruction
  - E.g., Add R1, 9 // R1 ← R1 + 9
- Direct
  - Operand directly read from memory, address is immediate
  - E.g., ADD R1, (9) //R1 ← R1 + M[9]
- Register
  - Operand is register content
  - E.g., ADD R1, R2 //R1 ← R1 + R2
- Register direct
  - Operand directly read from memory; address is register content
  - E.g., ADD R1, (R2) //R1 ← R1 + M[R2]
- Register indexed
  - Operand is an array element
  - E.g., ADD R1, R2, (9) //R1 ← R1 + M[R2 + 9]

# Types of ISA (1)

- Stack:
  - Arithmetic instructions have no explicitly declared operands
  - Operands are on stack inside CPU
  - E.g., ADD //Stack[top] ← Stack[top] + stack[top-1]
- Example: A = B * (C + D);
  - Requires converting statement into reverse polish notation
    - CD+B*=A
  - Reverse polish notation converted to assembly program
    - Program?
- Short instructions (advantage)
- Stack as LIFO buffer (disadvantage)



# Types of ISA (2)

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



# Types of ISA (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

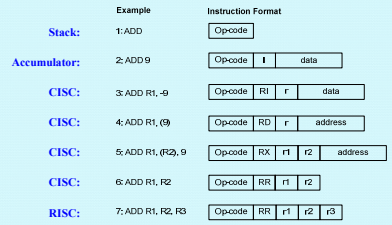


# Types of ISA (4)

- 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: A = B * (C + D);
    - Program?
  - Simpler and highly pipelined data path (advantage)
  - Requires compiler optimization to increase efficiency
  - The architecture of all modern processing cores



# Machine Instruction Format (Examples)






# 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 Program

```
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**



## Acc-ISA Example Assembly Program

```
.code
      LD  0
      ST  (sum)
      ST  (i)
L1:
      CMP    7
      JGTL2
      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
```

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



## Pentium IV Example Assembly Program (CISC)

**Note, arithmetic instructions can access memory**

```
      movl   $0, -48(%ebp)            //initialize sum, Memory[%ebp - 48] ← 0 (sum = 0)
      movl   $0, -44(%ebp)            //initialize i, Memory[%ebp - 44] ← 0 (I = 0)
L7:
      cmpl   $7, -44(%ebp)            //is i > 7?
      jg     L8                       //if yes, jump to L8
      movl   -44(%ebp), %eax          //get i, %eax ← Memory[i]
      movl   -40(%ebp,%eax,4), %edx   //get array[i], %edx ← Memory[array + i + 4]
      leal   -48(%ebp), %eax          //get address of sum, %eax ← %ebp - 48
      addl   %edx, (%eax)             //Memory[sum] ← array[i] + Memory[sum]
      leal   -44(%ebp), %eax          //get address of i, %eax ← %ebp - 44
      incl   (%eax)                   //Memory[i] ← Memory[i] + 1
      jmp    L7                       //repeat
L8:   ...
```



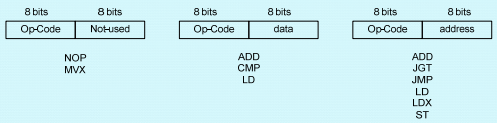
## Sparc Example Assembly Program

- **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:
```



## Acc-ISA Instruction format






## Acc-ISA Machine Instructions

| Address | Instruction | | Instruction in binary | In hex |
|---|---|---|---|---|
| 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: | … | | | |

# 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   $0x0,0xffffffd0(%ebp)
401347:       c7 45 d4 00 00 00 00  movl   $0x0,0xffffffd4(%ebp)
40134e:       83 7d d4 07           cmpl   $0x7,0xffffffd4(%ebp)
401352:       7f 13                 jg     401367 <_main+0x77>
401354:       8b 45 d4              mov    0xffffffd4(%ebp),%eax
401357:       8b 54 85 d8           mov    0xffffffd8(%ebp,%eax,4),%edx
40135b:       8d 45 d0              lea    0xffffffd0(%ebp),%eax
40135e:       01 10                 add    %edx,(%eax)
401360:       8d 45 d4              lea    0xffffffd4(%ebp),%eax
401363:       ff 00                 incl   (%eax)
401365:       eb e7                 jmp    40134e <_main+0x5e>
401367:       b8 02 00 00 00        ...
```



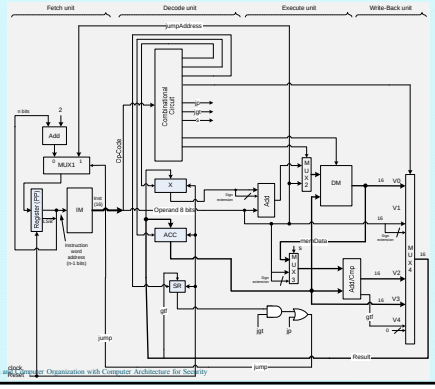
# AltraSparc II Machine instructions (RISC)

- Fixed-size instructions
- Requires simpler data path and control unit

```
...104f4:       c0 27 bf d0     clr  [ %fp + -48 ]
   104f8:       c0 27 bf d4     clr  [ %fp + -44 ]
   104fc:       c2 07 bf d4     ld  [ %fp + -44 ], %g1
   10500:       80 a0 60 07     cmp  %g1, 7
   10504:       14 80 00 0f     bg  10540 <main+0x90>
   10508:       01 00 00 00     nop
   1050c:       c2 07 bf d4     ld  [ %fp + -44 ], %g1
   10510:       85 28 60 02     sll  %g1, 2, %g2
   10514:       82 07 bf f8     add  %fp, -8, %g1
   10518:       82 00 80 01     add  %g2, %g1, %g1
   1051c:       c4 07 bf d0     ld  [ %fp + -48 ], %g2
   10520:       c2 00 7f e0     ld  [ %g1 + -32 ], %g1
   10524:       82 00 80 01     add  %g2, %g1, %g1
   10528:       c2 27 bf d0     st  %g1, [ %fp + -48 ]
   1052c:       c2 07 bf d4     ld  [ %fp + -44 ], %g1
   10530:       82 00 60 01     inc  %g1
   10534:       c2 27 bf d4     st  %g1, [ %fp + -44 ]
   10538:       10 bf ff f1     b  104fc <main+0x4c>
   1053c:       01 00 00 00     nop
   10540:       ...
```



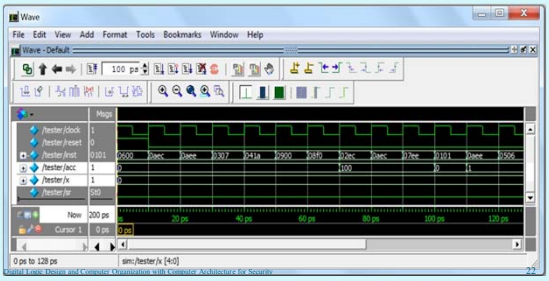
# Acc-ISA Data Path: Single-Cycle






# Acc-ISA Single Cycle Simulation

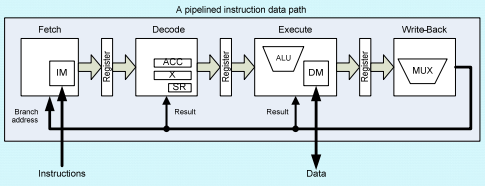
- Illustrates one instruction executed per cycle






# Acc-ISA Pipelined Data path

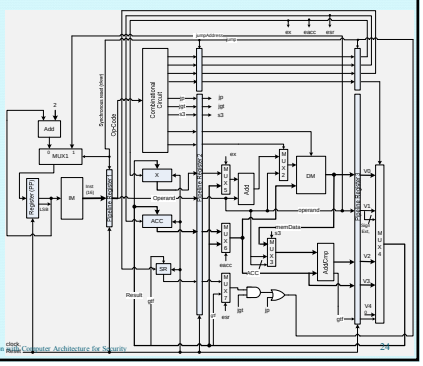
- Data path consists of four stages (block diagram)





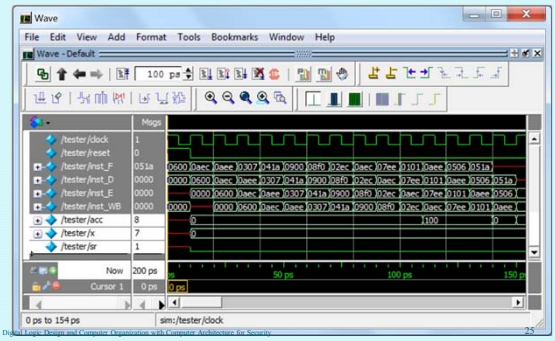

# Acc-ISA Pipelined Data Path

- Includes pipeline registers
- Also includes forwarding unit
- In this case, no need for a hazard unit (later)



# Acc-ISA Pipeline Simulation

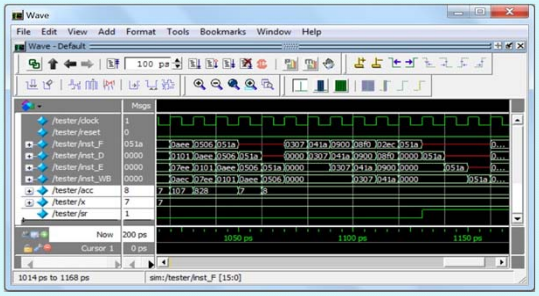
- Illustrating concurrent instruction execution






# Pipeline Simulation

- Illustrates pipeline flush on jumps






# How forwarding Unit Works

- Operands from write-back stage forwarded to execution unit

**With Forwarding Unit:**

No Hazard Unit is needed in this case

Arrow indicates Result forwarded from Write-Back stage to Execute stage

| | | | | | |
|---|---|---|---|---|---|
| Write-Back | | | | LD (i) | ADD 1 |
| Execute | | | LD (i) | ADD 1 | |
| Decode | | LD (i) | ADD 1 | | |
| Fetch | LD (i) | ADD 1 | | | |
| Clock Cycle: | 1 | 2 | 3 | 4 | 5 |

(a) With a forwarding unit

Bobble is used to indicate a pipeline stall

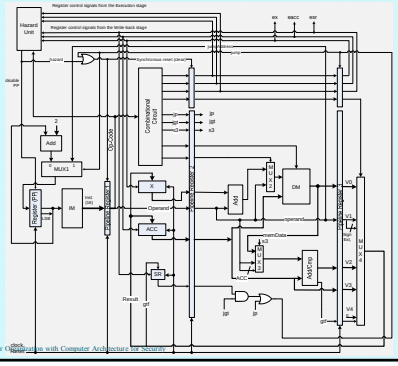
**Without Forwarding Unit:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Write-Back | | | | LD (i) | O | O | ADD 1 |
| Execute | | | LD (i) | O | O | ADD 1 | |
| Decode | | LD (i) | ADD 1 | ADD 1 | ADD 1 | | |
| Fetch | LD (i) | ADD 1 | | | | | |
| Clock Cycle: | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

(b) Without a forwarding unit but with a hazard unit



# Acc-ISA pipelined data Path with Hazard Unit





# Performance Parameters

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

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

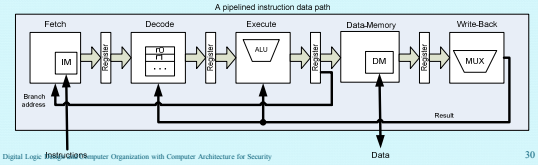
- Execution time (T) of a program

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



# RISC-ISA Data path

- Data memory in its own stage
  - Creates a one cycle delay to load data
    - Compiler optimization may hide the delay
  - Reduces execution stage propagation delay
    - Increases operating clock frequency



# RISC-ISA Example Program

```
.code
        ST      (sum), R0     //assume R0 is always 0
        ST      (i), R0
L1:
        LD      R1, (i)
        CMP     R1, 7
        JGT     L2
        LD      R2, (sum)
        LD      R3, R1, (array)
        ADD     R4, R2, R3
        ST      (sum), R4
        ADD     R1, R1, 1
        ST      (i), R1
        JMP     L1
L2:     Ix
        Iy
        Iz
.data
array:  RB      8
i:      RB      1
sum:    RB      1
```

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



# Pipeline Chart

- Illustrates "ADD R4, R2, R3" is delayed due to "LD R3, R1, (array)"
- After memory access, memory content is forwarded to execution unit

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Write-Back | . . . | . . . | . . . | . . . | LD R2, … | LD R3, … | O |
| Data Memory | . . . | . . . | . . . | LD R2, … | LD R3, … | O | ADD R4, R2, R3 |
| Execute | . . . | . . . | LD R2, … | LD R3, … | O | ADD R4, R2, R3 | . . . |
| Decode | . . . | LD R2, … | LD R3, … | ADD R4, R2, R3 | ADD R4, R2, R3 | . . . | . . . |
| Fetch | LD R2, … | LD R3, … | ADD R4, R2, R3 | . . . | . . . | . . . | . . . |



# RISC Compiler Optimization

- When possible moves an instruction to between LD and dependent next instruction
- Example
  - Before:

```
LD      R3, R1, (array)
ADD     R4, R2, R3
ST      (sum), R4
ADD     R1, R1, 1
```

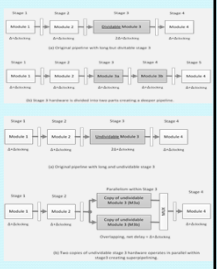
  - After:

```
LD      R3, R1, (array)
ADD     R1, R1, 1
ADD     R4, R2, R3
ST      (sum), R4
```



# Deep Pipelining

- Advantage:
  - Creates pipeline with more stages
    - By physically dividing stages into two more stages (when possible)
    - By superpipelining
  - Increases clock frequency (decreases τ)
    - Reduces execution time
- Disadvantage:
  - More cycles lost due to pipeline flush (due to branching)
  - More complex forwarding and hazard units
  - Increase CPI
- Other techniques to reduce CPI:
  - Branch prediction to minimize pipeline flushing overhead





# What is Branch Prediction?

- Early fetch of target instruction
- Static prediction
  - Uses a set of fixed rules
    - During compilation
      - E.g., forward branching executes the "else" part, no branching executes the "then" part
    - During execution
      - Prediction is taken when branching backward and not taken when branching forward
      - Ideal for for-loop and do-while
    - Prediction logic in the decode stage
- Dynamic prediction
  - Uses table to track history of each branch or multiple branches
  - Prediction logic in the fetch unit



# Dynamic Prediction Logic

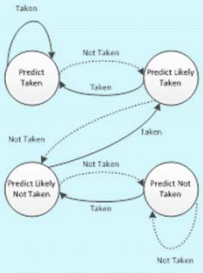
- 2-bit predictor
  - Works well with nested for-loops

A. Local predictor
  - Use 2-bit predictor per branch instruction

B. Global predictor
  - Use 2-bit predictor for each of k previous branch instructions

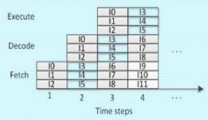
C. Correlation predictor
  - Combination of A and B

D. Tournament predictor
  - Implement both A and B and use a 2-bit predictor to choose A or B



# Other ways to increase CPU Throughput (1)

- **ILP: Superscalar processor**
  - **Statically determine ILP**
    - **Done in software by compiler**
    - **E.g., Intel Itanium, ARM Cortex-A8**
    - **Advantages**
      - Less power consumption
      - Good for hand-held devices
    - **Disadvantages**
      - Programs must be re-compiled
  - **Dynamically determined ILP**
    - **Done in hardware**
      - E.g., Intel Core i7
    - **Disadvantage**
      - More complex
        - Need to resolve WAR and WAW hazards to increase performance
      - More hardware, more power consumption, more heat generated
    - **Advantages**
      - Exact program execution is known during run time
      - WAR and WAW hazards resolved by register renaming
      - More chances for ILP






# Other ways to increase CPU Throughput (2)

- **Multithreading**
  - **A. Switch to another thread when one thread causes cache miss**
    - **Simple, less hardware (advantage)**
    - **Thread execution may be delayed (disadvantage)**
    - **Pipeline flush on every switch (disadvantage)**
  - **B. Switch threads on every cycle and on cache miss**
    - **More even execution of threads**
    - **Threads would be waiting for their turn (disadvantage)**
  - **C. Simultaneously executing instructions from the threads**
    - **Enables thread-level parallelism (TLP)**
- **CPU executes multiple (e.g., 2) threads concurrently**
- **Increases overall IPC, not IPC of each program**
- **Improves system performance**