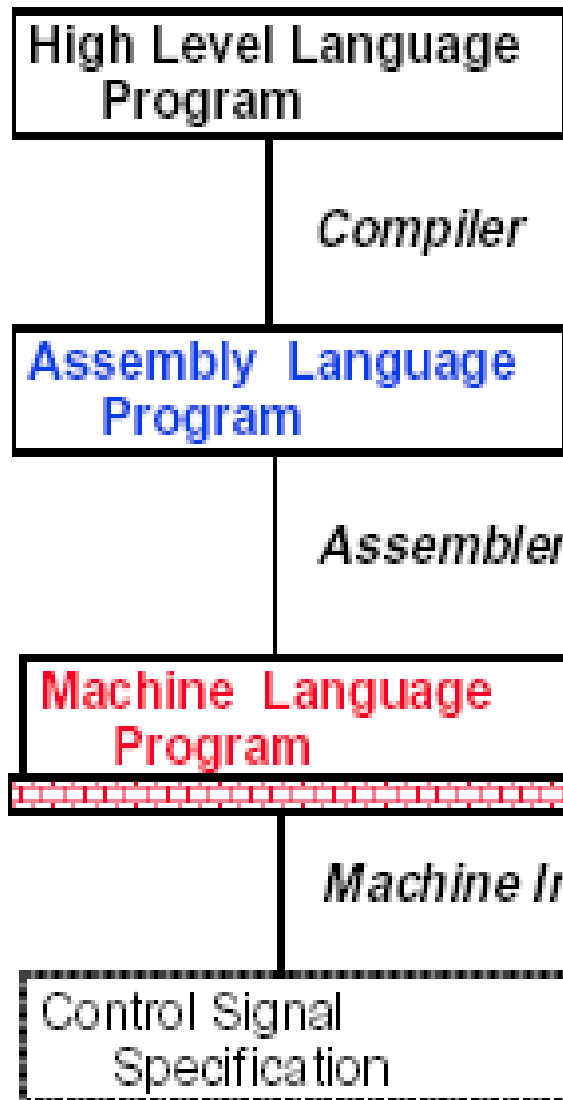# COMPUTER HARDWARE

Instruction Set Architecture

Overview

l Computer architecture

l Operand addressing

- Addressing architecture

- Addressing modes

l Elementary instructions

- Data transfer instructions

- Data manipulation instructions

  ◆ Floating point computations

- Program control instructions

  ◆ Program interrupt and exceptions

Overview

High Level Language Program

— Compiler →

Assembly Language Program

— Assembler →

Machine Language Program

— Machine Interpretation →

Control Signal Specification

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

```
lw$15, 0($2)
lw$16, 4($2)
sw     $16, 0($2)
sw     $15, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:11] & MASK

Computer Architecture

- Instruction set architecture
  - A set of hardware-implemented instructions, the symbolic name and the binary code format of each instruction
- Organization
  - Structures such as datapath, control units, memories, and the busses that interconnect them
- Hardware
  - The logic, the electronic technology employed, the various physical design aspects of the computer
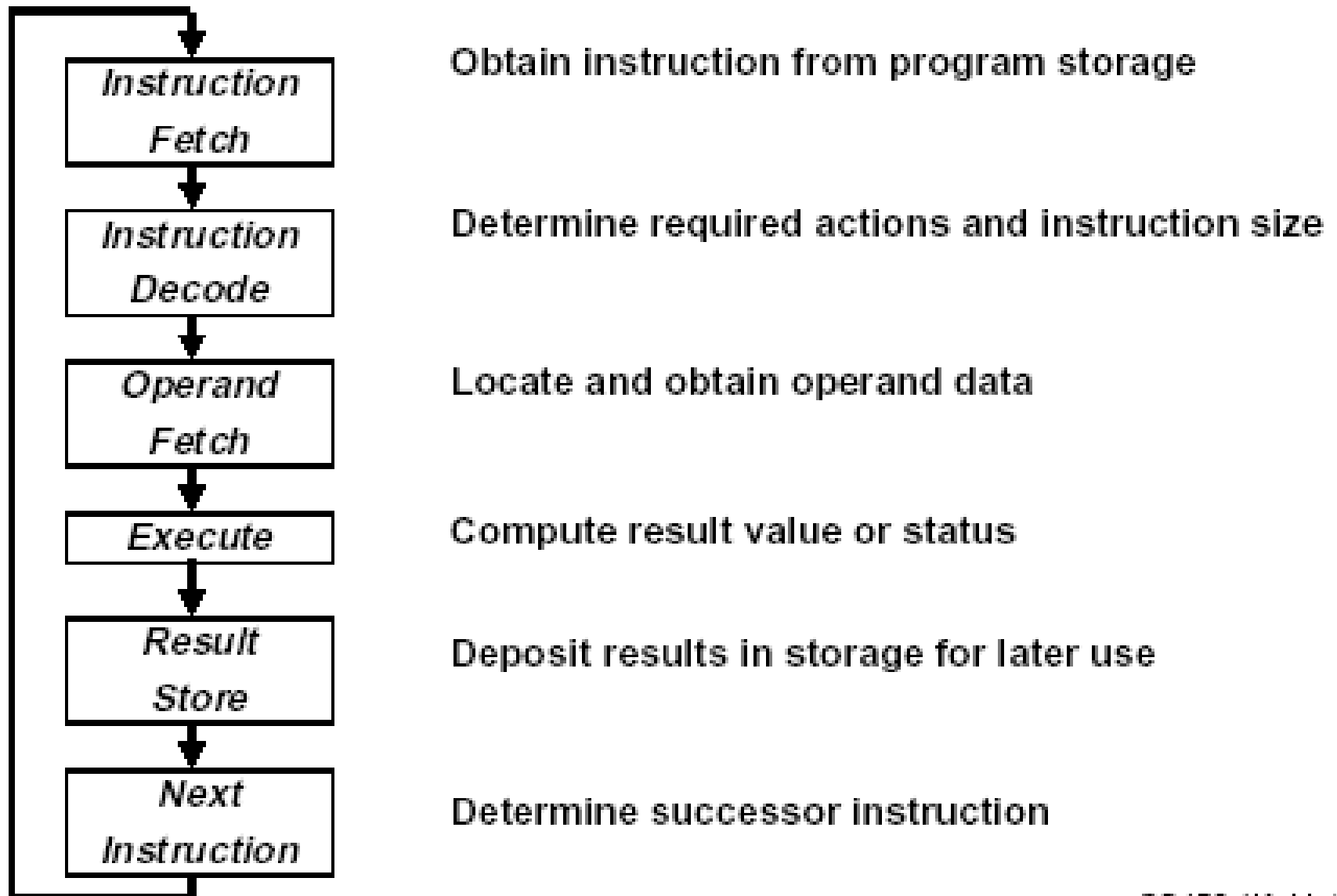
Example ISAs (Instruction Set Architectures)

- RISC (Reduced Instruction Set Computer)
  - Digital Alpha
  - Sun Sparc
  - MIPS RX000
  - IBM PowerPC
  - HP PA/RISC
- CISC (Complex Instruction Set Computer)
  - Intel x86
  - Motorola 68000
  - DEC VAX
- VLIW (Very Large Instruction Word)
  - Intel Itanium

Instruction Set Architecture

l A processor is specified completely by its ***instruction set architecture (ISA)***

l Each ISA will have a variety of instructions and instruction formats, which will be interpreted by the processor's control unit and executed in the processor's datapath

l An instruction represents the smallest indivisible unit of computation. It is a string of bits grouped into different numbers and size of ***substrings*** (***field***s)

– ***Operation code (opcode):*** *the operation to be performed*

– ***Address field:*** *where we can find the operands needed for that operation*

– ***Mode field:*** *how to derive the data's effective address from the information given in the address field*

– ***Other fields:*** *constant immediate operand or shift*

Computer Operation Cycle

Instruction Fetch	Obtain instruction from program storage
Instruction Decode	Determine required actions and instruction size
Operand Fetch	Locate and obtain operand data
Execute	Compute result value or status
Result Store	Deposit results in storage for later use
Next Instruction	Determine successor instruction

CS152 (Kubi)

Pipelined Datapath

Conventional:

A maximum of 0.8 ns (0.6+ 0.2) is required to read two operands from the register file or to read one operand from the register file and obtain a constant from MUX B.

A maximum of 0.8 ns is required to execute an operation in the functional unit. Also, a maximum of 0.8 ns is required to write the result back into the register file, including the delay of MUX D.

Adding these delays, we find that 2.4 ns (i.e., 416.7 MHz) are required to perform a single microoperation.

Pipelined:

Three sets of registers break the delay of the original datapath into three parts. These registers are shown crosshatched in blue. The register file contains the first set of registers. Cross-hatching covers only the top half of the register file, since the lower half is viewed as the combinational logic that selects the two registers to be read. The two registers that store the *A* data from the register file and the output of MUX B constitute the second set of registers. The third set of registers stores the inputs to MUX D.
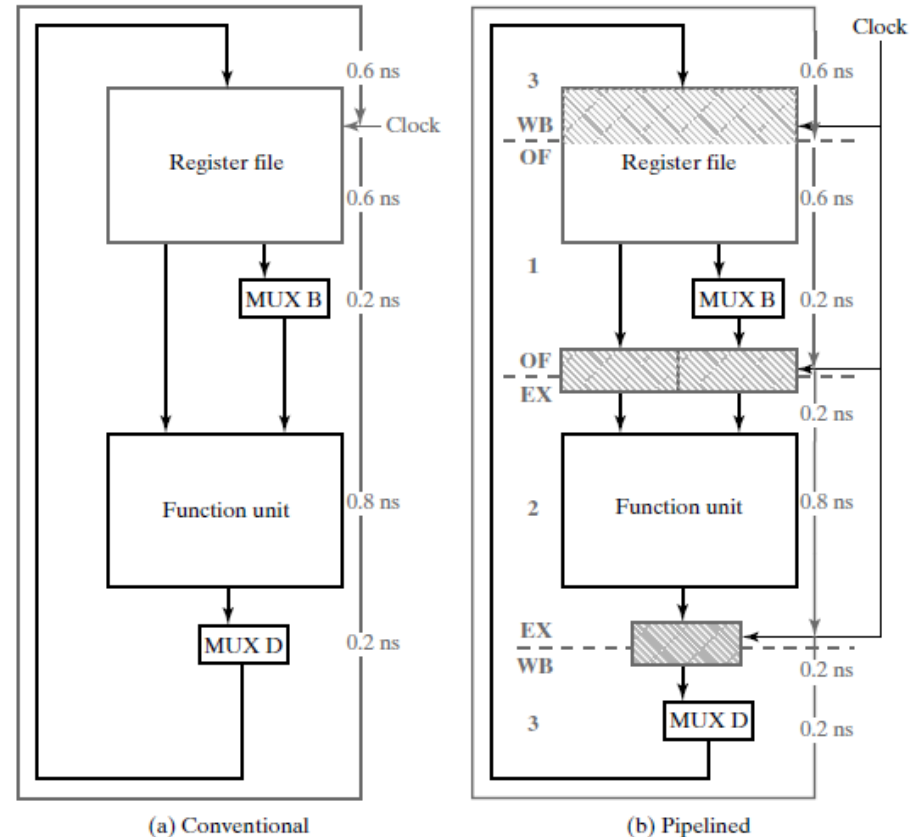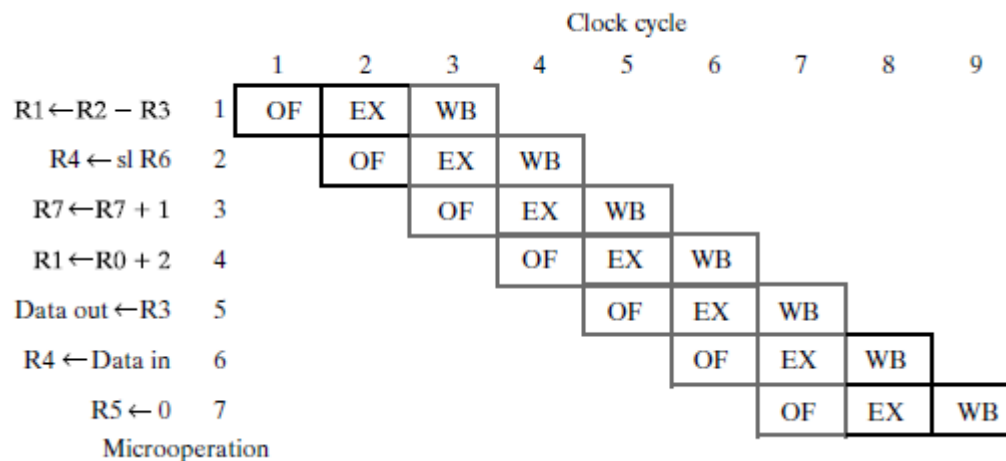
Execution of Pipeline Microoperations

pipeline execution pattern diagram:

Each vertical position in this diagram represents a microoperation to be performed, and each horizontal position represents a clock cycle. An entry in the diagram represents the stage of processing of the microoperation.

So, for example, the execution (EX) stage of microoperation 4, which adds the constant 2 to R0, occurs in clock cycle 5.

We can see from the overall diagram that the sequence of seven microoperations requires nine clock cycles to execute completely. The time required for execution is $9 \times 1 = 9$ ns, compared to $7 \times 2.4 = 16.8$ ns for the conventional datapath. Thus, the sequence of microoperations is executed about 1.9 times faster using the pipeline.

		Clock cycle								
		1	2	3	4	5	6	7	8	9
R1←R2 − R3	1	OF	EX	WB						
R4 ← sl R6	2		OF	EX	WB					
R7←R7 + 1	3			OF	EX	WB				
R1←R0 + 2	4				OF	EX	WB			
Data out ←R3	5					OF	EX	WB		
R4 ←Data in	6						OF	EX	WB	
R5 ← 0	7							OF	EX	WB
Microoperation										

Example for Pipelined Datapath Unit

A pipelined datapath is similar to that in Figure 1(b), but with the delays from the top to the bottom replaced by the following values: 0.5 ns, 0.5 ns, 0.1 ns, 0.1 ns, 0.7 ns, 0.1 ns, and 0.1 ns.

Determine

(a) the maximum clock frequency

(b) the latency time, and (c) the maximum throughput for this datapath.
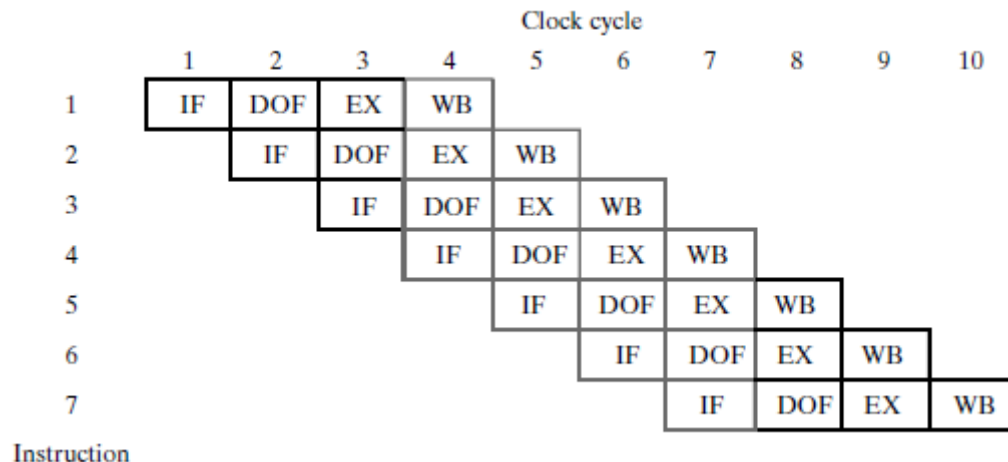
Pipelined Control

The added first stage is the instruction fetch stage, denoted by IF, which lies wholly in the control. In this stage, the instruction is fetched from the instruction memory, and the value in the *PC* is updated.

In the second stage, DOF (decode and operand fetch), decoding of the IR into control signals takes place. Among the decoded signals, the register file addresses AA and BA and the multiplexer control signal MB are used in this stage for operand fetch.

At the end of the fourth clock period, instruction 1 has completed execution, instruction 2 is three-fourths finished, instruction 3 is half finished, and instruction 4 is one-fourth finished.

We can see from the overall diagram that the complete program of seven instructions requires 10 clock cycles to execute. Thus, the time required is 10 ns, compared to 23.8 ns (7x3.4ns) for the single-cycle computer, and the program is executed about 2.4 times faster.

						Clock cycle				
	1	2	3	4	5	6	7	8	9	10
1	IF	DOF	EX	WB						
2		IF	DOF	EX	WB					
3			IF	DOF	EX	WB				
4				IF	DOF	EX	WB			
5					IF	DOF	EX	WB		
6						IF	DOF	EX	WB	
7							IF	DOF	EX	WB

Instruction

Register Set

- Programmer accessible registers (R0 to R7 in previous multi-cycle computer)

- Other registers

  - Registers in the register file accessible only to microprograms (R8 to R15)

  - Instruction registers (IR)

  - Program counter (PC)

  - Pipeline registers

  - Processor status register (PSR: CVNZ state)

  - Stack pointer (SP)

Operand Addressing

- Operand: register value, memory content, or immediate

- *Explicit address*: address field in the instruction

- *Implied address*: the location of operand is specified by the opcode or other operand address

Three Address Instructions

l Example: X=(A+B)(C+D)

l Operands are in memory address symbolized by the letters A,B,C,D, result stored memory address of X

ADD T1, A, B	M[T1] ← M[A] + M[B]
ADD T2, C, D	M[T2] ← M[C] + M[D]
MUX X, T1, T2	M[X] ← M[T1] X M[T2]

OR

ADD R1, A, B	R1 ← M[A] + M[B]
ADD R2, C, D	R2 ← M[C] + M[D]
MUX X, R1, R2	M[X]← R1 X R2

l +: Short program, 3 instructions

l -: Binary coded instruction require more bits to specify three addresses

Two Address Instructions

l The first operand address also serves as the implied address for the result

MOVE T1, A	M[T1] ← M[A]
ADD T1, B	M[T1] ← M[T1] + M[B]
MOVE X, C	M[X] ← M[C]
ADD X, D	M[X] ← M[X] + M[D]
MUX X, T1	M[X] ← M[X] X M[T1]

l 5 instructions

One Address Instructions

l Implied address: a register called *an accumulator ACC* for one operand and the result, *single-accumulator architecture*

LD	A	ACC ← M[A]
ADD	B	ACC ← ACC + M[B]
ST	X	M[X] ← ACC
LD	C	ACC ← M[C]
ADD	D	ACC ← ACC + M[D]
MUX	X	ACC ← ACC X M[X]
ST	X	M[X] ← ACC

7 instructions

l All operations are between the ACC register and a memory operand

Zero Address Instructions

- Use stack (FILO):

  - ADD $TOS \leftarrow TOS + TOS_{-1}$
  - PUSH X $TOS \leftarrow M[X]$
  - POP   X $M[X] \leftarrow TOS$

PUSH A	$TOS \leftarrow M[A]$	
PUSH B	$TOS \leftarrow M[B]$	
ADD	$TOS \leftarrow TOS + TOS_{-1}$	
PUSH C	$TOS \leftarrow M[C]$	8 instructions
PUSH D	$TOS \leftarrow M[D]$	
ADD	$TOS \leftarrow TOS + TOS_{-1}$	
MUX	$TOS \leftarrow TOS \times TOS_{-1}$	
POP X	$M[X] \leftarrow TOS$	

- Data manipulation operations: between the stack elements
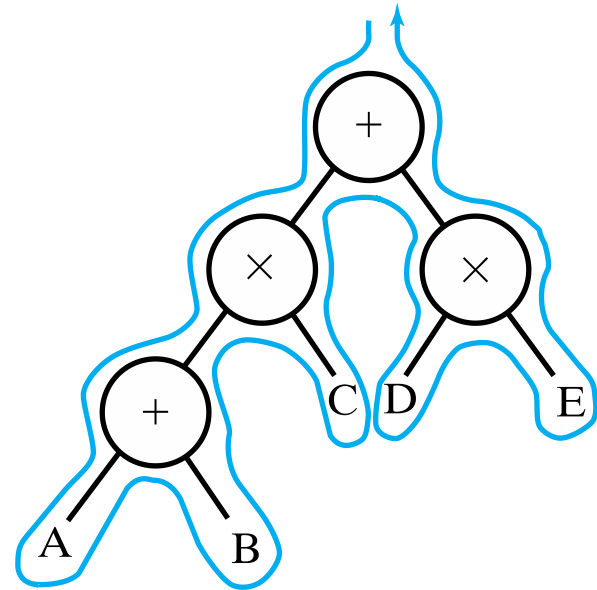
- Transfer operations: between the stack and the memory

Stack Instructions

Push: SP←SP-1; TOS←R1

Pop: R1←TOS; SP←SP +1
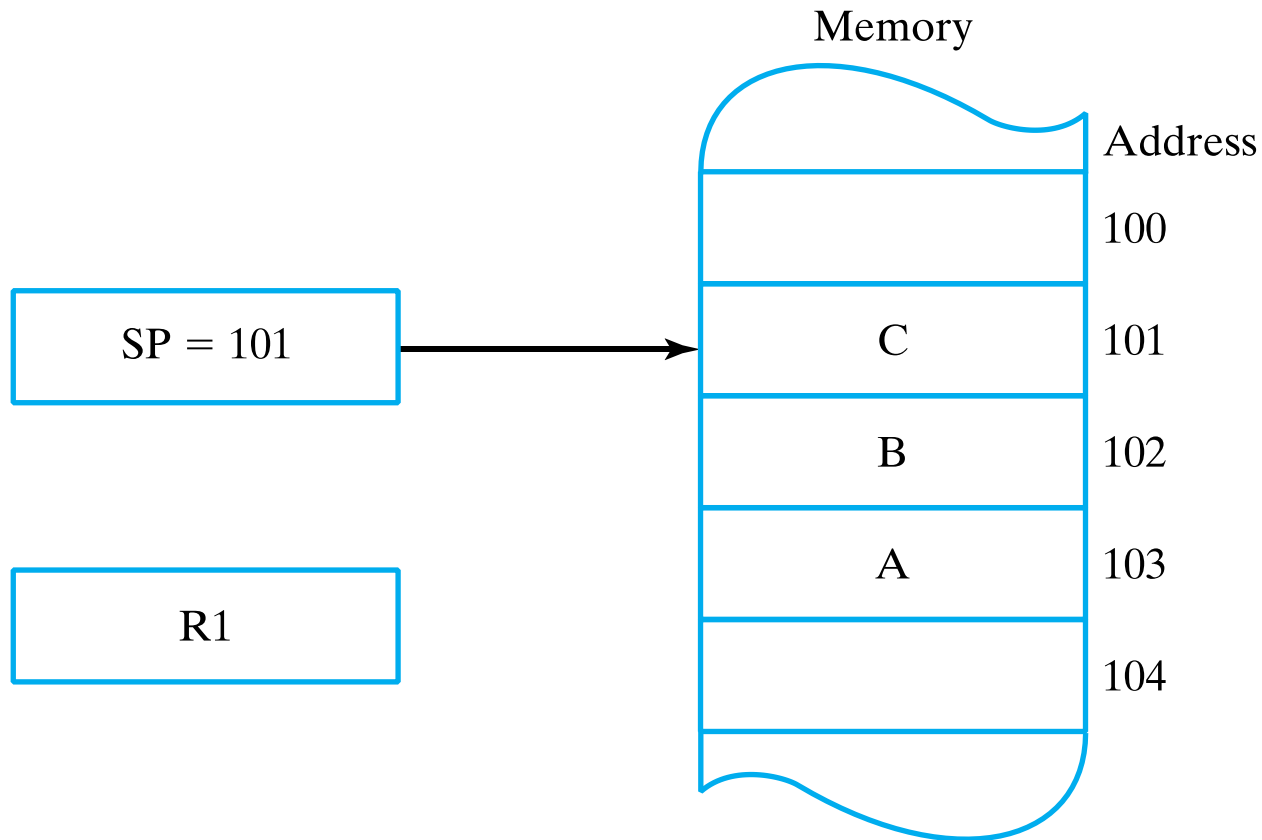
Stack Architecture

- The infix architecture
  (A + B) x C + (D x E)
- Reverse Polish Notation (RPN)
  A B + C x D E x +

A

B
A

A + B

C
A + B

$(A + B) \times C$

D
$(A + B) \times C$

E
D
$(A + B) \times C$

$D \times E$
$(A + B) \times C$

$(A + B) \times C + D \times E$

Addressing Architecture

l Defines:

– Restriction on the number of memory addresses in instructions

– Number of operands

l Two kinds of addressing architecture:

– Memory-to-memory architecture

◆ Only one register - PC

◆ All operands from memory, and results to memory

◆ Many memory accesses

– Register-to-register (load/store) architecture

◆ Restrict only one memory address to load/store types, all other operations are between registers

```
LD    R1, A          R1 ← M[A]
LD    R2, B          R2 ← M[B]
ADD  R3, R1, R2      R3 ← R1 + R2
LD    R1, C          R1 ← M[C]
LD    R2, D          R2 ← M[D]
ADD  R1, R1, R2      R1 ← R1 + R2
MUL  R1, R1, R3      R1 ← R1 X R3
ST    X, R1          M[X] ← R1
```

Addressing Modes

- **Address field:** contains the information needed to determine the location of the operands and the result of an operation

- **Addressing mode:** specifies how to interpret the information within this address field, how to compute the **actual** or **effective** address of the data needed.

- Availability of a variety of addressing modes lets programmers write more efficient code
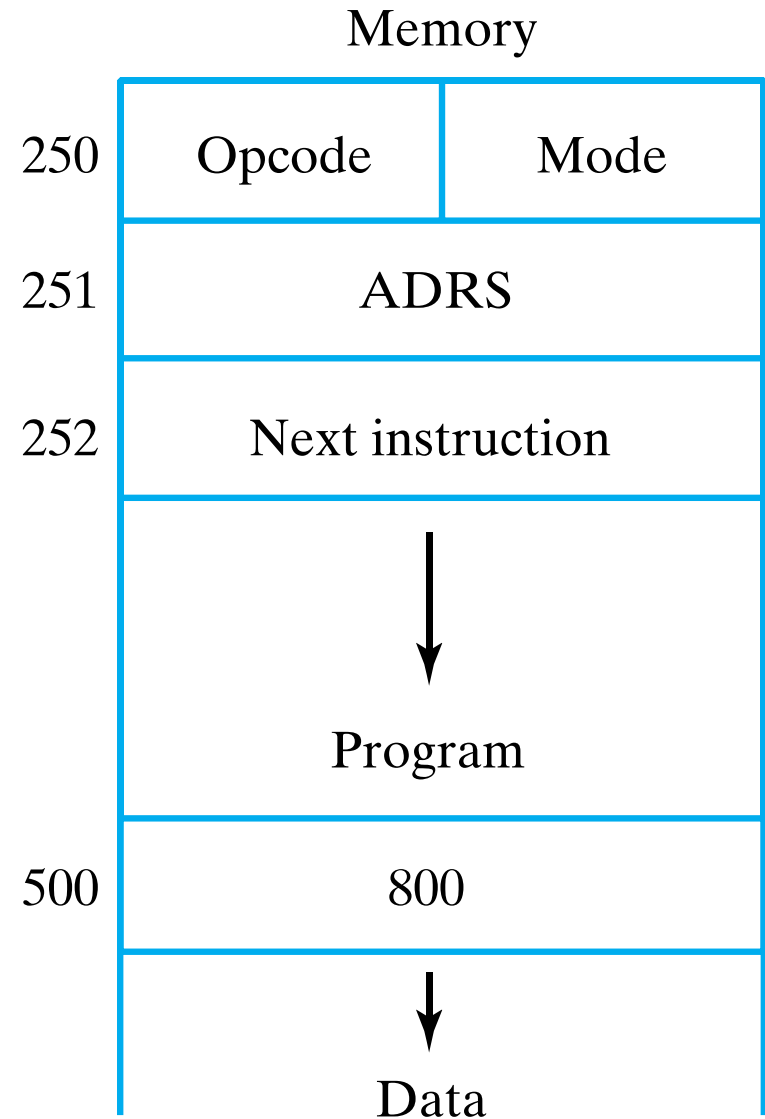
Addressing Modes

- **Implied** mode - implied in the opcode, such as stack, accumulator
- **Immediate** mode (operand) - a = 0x0801234
- **Register** mode - a=R[b]
- **Register-indirect** mode – a =M[R[b]]
- **Direct** addressing mode - a = M[0x0013df8]
- **Indirect** Addressing mode - a= M[M[0x0013df8]]
- **PC-relative** addressing – branch etc. (offset + PC)
- **Indexed** addressing - a=b[1]

Demonstrating Direct Addressing

Memory

PC = 250

ACC

250	Opcode / Mode
251	ADRS
252	Next instruction
	Program
500	800
	Data

Opcode: Load ACC
Mode: Direct address
ADRS: 500
Operation: ACC ← 800

Example

Opcode: Load to ACC,   ADRS or NBR=500

Memory

250	Opcode / Mode
251	ADRS or NBR = 500
252	Next instruction
400	700
500	800
752	600
800	300
900	200

PC = 250	R1 = 400	ACC

Refers to Figure 10-6

Addressing Mode	Symbolic Convention	Register Transfer	Effective Address	Contents of *ACC*
Direct	LDA ADRS	$ACC \leftarrow M[ADRS]$	500	800
Immediate	LDA #NBR	$ACC \leftarrow NBR$	251	500
Indirect	LDA [ADRS]	$ACC \leftarrow M[M[ADRS]]$	800	300
Relative	LDA $ADRS	$ACC \leftarrow M[ADRS + PC]$	752	600
Index	LDA ADRS (R1)	$ACC \leftarrow M[ADRS + R1]$	900	200
Register	LDA R1	$ACC \leftarrow R1$	—	400
Register-indirect	LDA (R1)	$ACC \leftarrow M[R1]$	400	700

Instruction Set Architecture

	RISC (reduced instruction set computers)	CISC (complex instruction set computers)
Memory access	restricted to load/store instructions, and data manipulation instructions are register-to-register	is directly available to most types of instructions
Addressing mode	limited in number	substantial in number
Instruction formats	all of the same length	of different lengths
Instructions	perform elementary operations	perform both elementary and complex operations
Control unit	Hardwired, high throughput and fast execution	Microprogrammed, facilitate compact programs and conserve memory,

Data Transfer Instructions

l Data transfer: memory$\leftarrow$ $\rightarrow$registers, processor registers $\leftarrow$ $\rightarrow$ input/output registers, among the processor registers

l Data transfer instructions

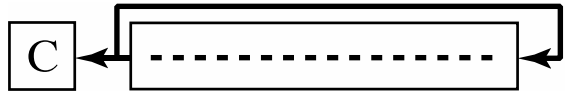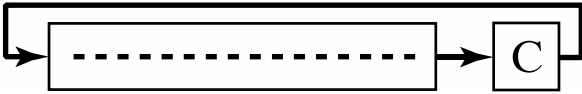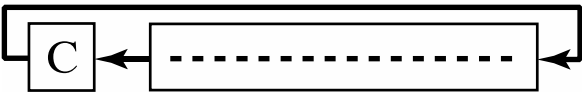| Name	Mnemonic
Load	LD
Store	ST
Move	MOVE
Exchange	XCH
Push	PUSH
Pop	POP
Input	IN
Output	OUT
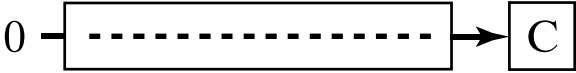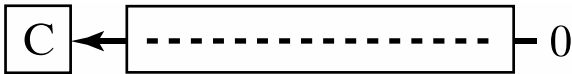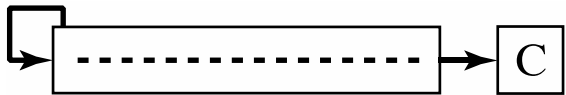
I/O

- Input and output (I/O) instructions transfer data between processor registers and I/O devices
  - Ports

- Independent I/O system: address range assigned to memory and I/O ports are independent from each other

- Memory-mapped I/O system: assign a subrange of the memory addresses for addressing I/O ports

Data Manipulation Instructions

Arithmetic		Logical and bit manipulation		Shift instructions	
Name	**Mnemonic**	**Name**	**Mnemonic**	**Name**	**Mnemonic**
Increment	INC	Clear	CLR	Logical shift right	SHR
Decrement	DEC	Set	SET	Logical shift left	SHL
Add	ADD	Complement	NOT	Arithmetic shift right	SHRA
Subtract	SUB	AND	AND	Arithmetic shift left	SHRL
Multiply	MUL	OR	OR	Rotate right	ROR
Divide	DIV	Exclusive-OR	XOR	Rotate left	ROL
Add with carry	ADDC	Clear carry	CLRC	Rotate right with carry	RORC
Subtract with borrow	SUBB	Set Carry	SETC	Rotate left with carry	ROLC
Subtract reverse	SUBR	Complement carry	COMC		
Negate	NEG				

Typical Shift Instructions

Name	Mnemonic	Diagram
Logical shift right	SHR	
Logical shift left	SHL	
Arithmetic shift right	SHRA	
Arithmetic shift left	SHLA	
Rotate right	ROR	
Rotate left	ROL	
Rotate right with carry	RORC	
Rotate left with carry	ROLC	

Arithmetic Shift Left Operation Example

Convert the following decimal numbers to signed binary with eight bits each. Perform the arithmetic shift left and right operations and indicate the overflow(V) bit.

1. $(+62)_{10}=(.....)_2$ $(+62)_{10}=(0011\ 1110)_2$

❑ After Arithmetic Shift Left Operation, result will be $(0111\ 1100)_2=(+124)_{10}$

What about V(Overflow) bit ??? $(+62)_{10}=(00$11 1110$)_2$ $V=0\oplus0=0$

❑ After Arithmetic Shift Right Operation, result will be $(0001\ 1111)_2=(+31)_{10}$

What about V(Overflow) bit ??? No Overflow for

2. $(-75)_{10}=(.....)_2$ $(-75)_{10}=(1011\ 0101)_2$

Program Control Instructions

l Control over the flow of program execution and a capability of branching to different program segments

l One-address instruction:

- Jump: direct addressing
- Branch: relative addressing

Name	Mnemonic
Branch	BR
Jump	JMP
Call procedure	CALL
Return from procedure	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TEST

Conditional Branching Instructions

l May or may not cause a transfer of control, depending on the value of stored bits in the *PSR (processor state register)*

Branch Condition	Mnemonic	Test Condition
Branch if zero	BZ	$Z = 1$
Branch if not zero	BNZ	$Z = 0$
Branch if carry	BC	$C = 1$
Branch if no carry	BNC	$C = 0$
Branch if minus	BN	$N = 1$
Branch if plus	BNN	$N = 0$
Branch if overflow	BV	$V = 1$
Branch if no overflow	BNV	$V = 0$

Conditional Branching Instructions

- ## for Unsigned Numbers

Branch Condition	Mnemonic	Condition	Status Bits*
Branch if above	BA	$A > B$	$C + Z = 0$
Branch if above or equal	BAE	$A \geq B$	$C = 0$
Branch if below	BB	$A < B$	$C = 1$
Branch if below or equal	BBE	$A \leq B$	$C + Z = 1$
Branch if equal	BE	$A = B$	$Z = 1$
Branch if not equal	BNE	$A \neq B$	$Z = 0$

*Note that C here is a borrow bit.

- For arithmetic operations on Unsigned Number only C (Carry) and Z (Zero) bits are evaluated !!!

- It is assumed that a previous instruction has updated status bits C and Z after a subtraction A-B or some other similar instruction.

Example for branch condition on unsigned numbers

The program in a computer compares two unsigned numbers A and B by performing a subtraction $A - B$ and updating the status bits.

For operands let A =01011101 and B = 01011100,

(a) Evaluate the difference and interpret the binary result.

(b) Determine the values of status bits C (borrow) and Z (zero).

(c) List the conditional branch instructions from previous slide that will have a true condition.

Conditional Branching Instructions

- ## for Signed Numbers

Branch condition	Mnemonic	Condition	Status Bits
Branch if greater	BG	$A > B$	$(N \oplus V) + Z = 0$
Branch if greater or equal	BGE	$A \geq B$	$N \oplus V = 0$
Branch if less	BL	$A < B$	$N \oplus V = 1$
Branch if less or equal	BLE	$A \leq B$	$(N \oplus V) + Z = 1$
Branch if equal	BE	$A = B$	$Z = 1$
Branch if not equal	BNE	$A \neq B$	$Z = 0$

- For arithmetic operations on Signed Number only N (Negate), V (Overflow) and Z (Zero) bits are evaluated !!!

- It is assumed that a previous instruction has updated status bits N, V and Z after a subtraction A-B.

N	V	N⊕V	Evaluation
0	0	0	A>B
0	1	1	A<B
1	0	1	A<B
1	1	0	A>B

Example for branch condition on signed numbers

Consider the two 8-bit numbers $A = 10110110$ and $B = 00110111$.

(a) Give the decimal equivalent of each number, assuming that they are signed 2s complement.

(b) Add the two binary numbers and interpret the sum, assuming that the numbers are signed 2s complement.

(c) Determine the values of the C (carry), Z (zero), N (sign), and V (overflow) status bits after the additions.

(d) List the conditional branch instructions from Table on previous slide that will have a true condition for each addition.

Procedure Call and Return Instructions

❑ Procedure: self-contained sequence of instructions that performs a given computational task

❑ Call procedure instruction: one-address field

  ▪ Stores the value of the PC (return address) in a temporary location

  ▪ The address in the call procedure instruction is loaded into the PC

❑ Final instruction in every procedure: return instruction

  ▪ Take the return address and load into the PC

❑ Temporary Location: fixed memory location, processor register or memory stack

  ▪ E.g. stack

    ▪ Procedure call: SP←SP-1;M[SP] ←PC+4;PC←Effective address

    ▪ Return: PC ←M[SP]; SP ←SP+1

Interprets

- Types of Interrupts
    1. **External:** Hard Drive, Mouse, Keyboard, Modem, Printer

    2. **Internal :** Overflow; Divide by zero; Invalid opcode; Memory stack overflow; Protection violation

    3. **Software:** A software interrupt provides a way to call the interrupt routines normally associated with external or internal interrupts by inserting an instruction into the code.

Processing External Interrupts

- *SP ← SP -1*   Decrement stack pointer
- *M[SP] ← PC*   Store return address on stack
- *SP ← SP- 1*   Decrement stack pointer
- *M[SP] ← PSR*   Store processor status word on stack
- *EI ← 0*   Reset enable interrupt flip flop
- *INTACK ← 1*   Enable interrupt acknowledge
- *PC ← IVAD*   Transfer interrupt vector address to PC and Go to fetch