COMPUTER ARCHITECTURE & ORGANIZATION

(6CS4-04)

Unit-3

Central Processing Unit

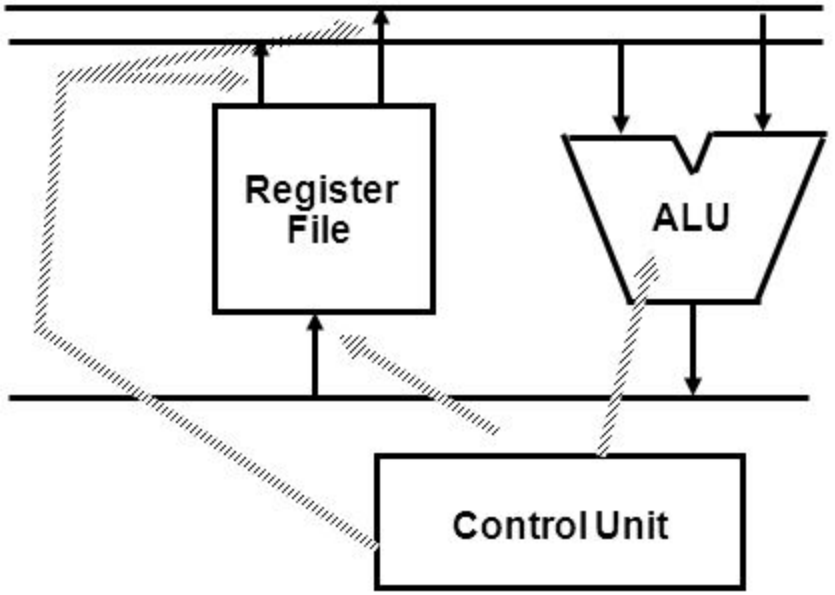
-Dr. Monica Lamba

CENTRAL PROCESSING UNIT

- Introduction
- General Register Organization
- Stack Organization
- Instruction Formats
- Addressing Modes
- Data Transfer and Manipulation
- Program Control
- Reduced Instruction Set Computer

MAJOR COMPONENTS OF CPU

- **Storage Components**
  - Registers
  - Flags
- **Execution (Processing) Components**
  - Arithmetic Logic Unit(ALU)
    - Arithmetic calculations, Logical computations, Shifts/Rotates
- **Transfer Components**
  - Bus
- **Control Components**
  - Control Unit

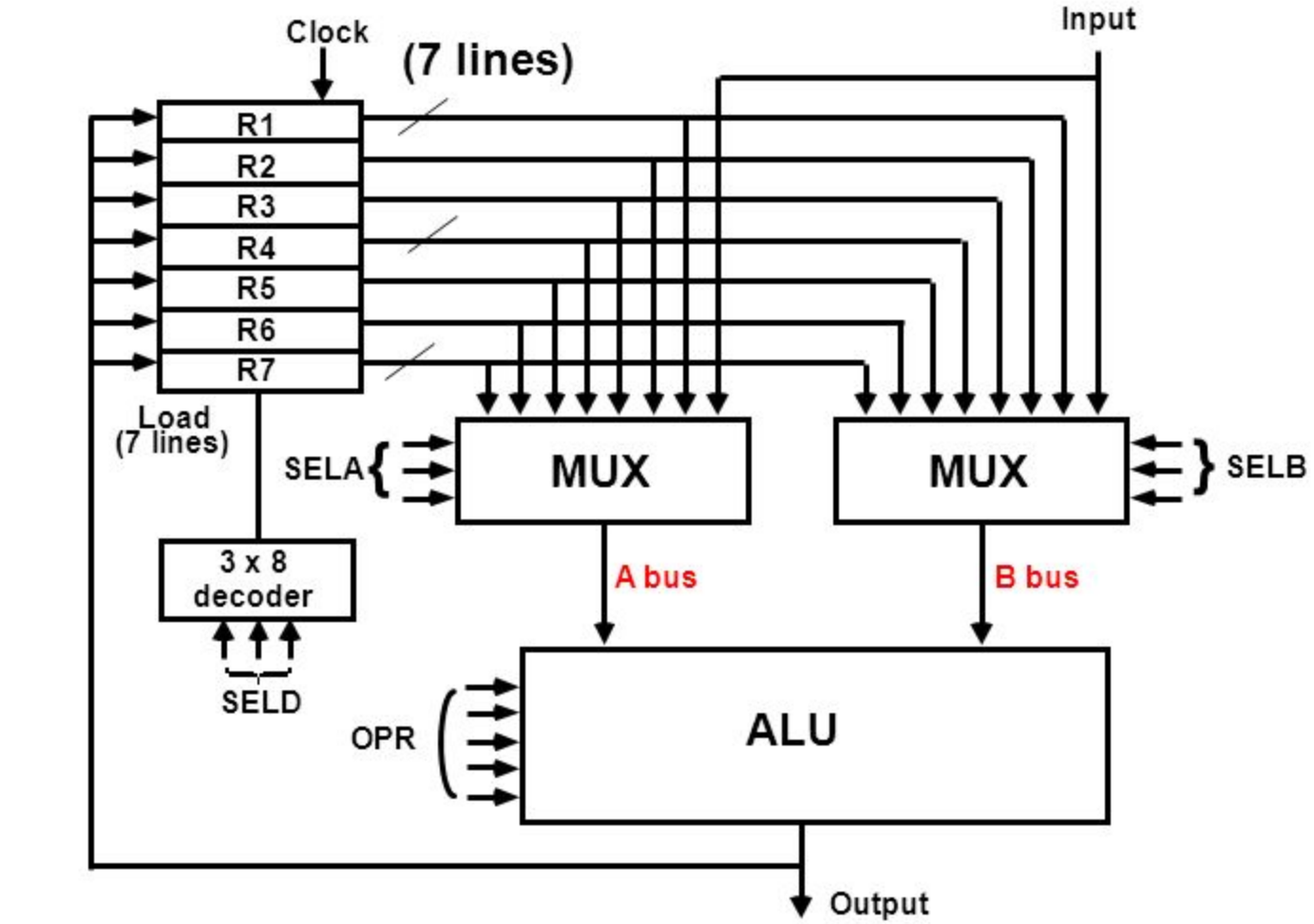


REGISTERS

- In Basic Computer, there is only one general purpose register, the Accumulator (AC)
- In modern CPUs, there are many general purpose registers
- It is advantageous to have many registers
  - Transfer between registers within the processor are relatively fast
  - Going "off the processor" to access memory is much slower

- How many registers will be the best ?

GENERAL REGISTER ORGANIZATION



OPERATION OF CONTROL UNIT

The control unit

Directs the information flow through ALU by

- Selecting various *Components* in the system
- Selecting the *Function* of ALU

Example: R1 ← R2 + R3

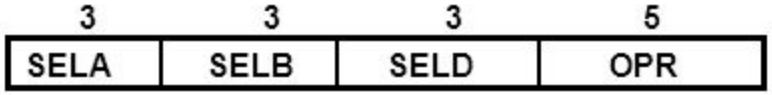
[1] MUX A selector (SELA): BUS A ← R2

[2] MUX B selector (SELB): BUS B ← R3

[3] ALU operation selector (OPR): ALU to ADD

[4] Decoder destination selector (SELD): R1 ← Out Bus

Control Word



Encoding of register selection fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

ALU CONTROL

Encoding of ALU operations

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	ADD A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Examples of ALU Microoperations

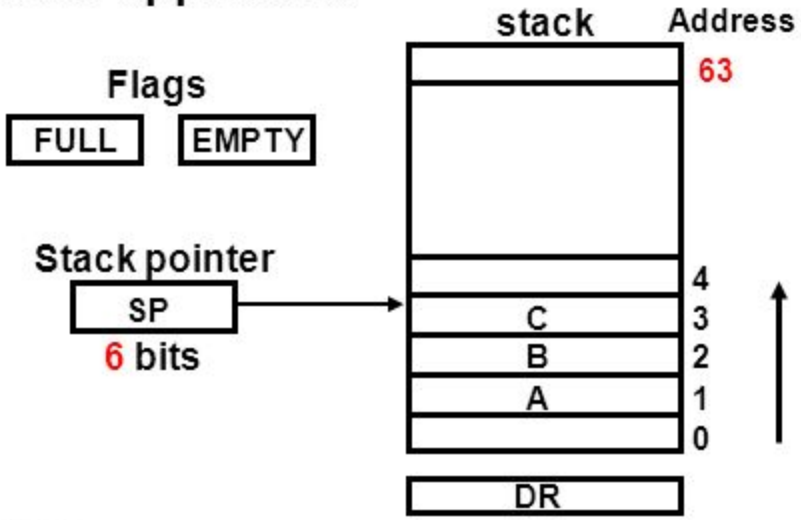
	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
R1 ← R2 – R3	R2	R3	R1	SUB	010 011 001 00101
R4 ← R4 ∨ R5	R4	R5	R4	OR	100 101 100 01010
R6 ← R6 + 1	R6	-	R6	INCA	110 000 110 00001
R7 ← R1	R1	-	R7	TSFA	001 000 111 00000
Output ← R2	R2	-	None	TSFA	010 000 000 00000
Output ← Input	Input	-	None	TSFA	000 000 000 00000
R4 ← shl R4	R4	-	R4	SHLA	100 000 100 11000
R5 ← 0	R5	R5	R5	XOR	101 101 101 01100

REGISTER STACK ORGANIZATION

Stack

- Very useful feature for nested subroutines, nested interrupt services
- Also efficient for arithmetic expression evaluation
- Storage which can be accessed in LIFO
- Pointer: SP
- Only PUSH and POP operations are applicable

Register Stack



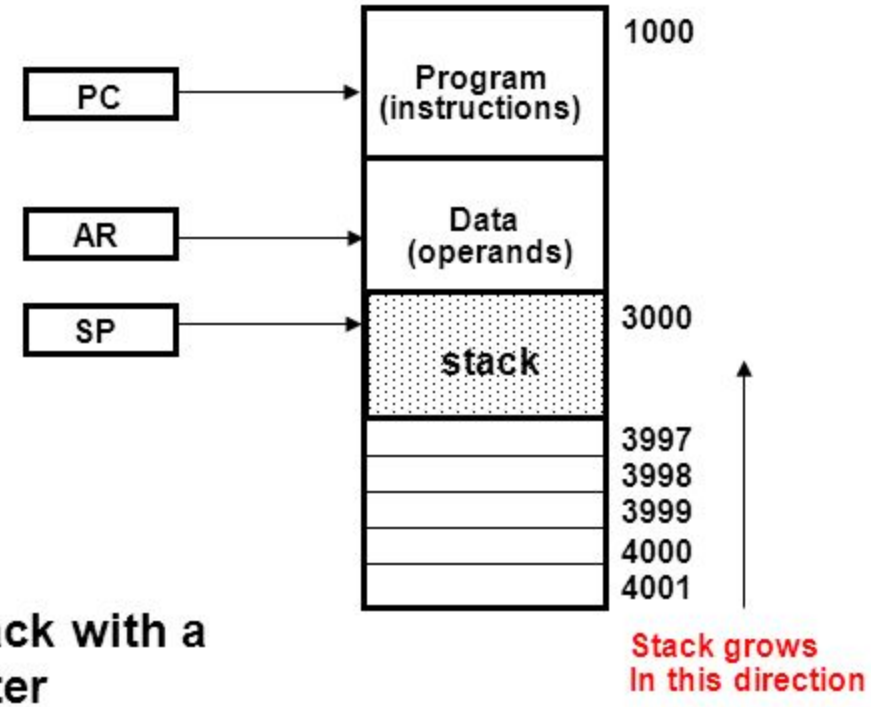
Push, Pop operations

/* Initially, SP = 0, EMPTY = 1, FULL = 0 */

PUSH	POP
SP ← SP + 1	DR ← M[SP]
M[SP] ← DR	SP ← SP – 1
If (SP = 0) then (FULL ← 1)	If (SP = 0) then (EMPTY ← 1)
EMPTY ← 0	FULL ← 0

MEMORY STACK ORGANIZATION

Memory with Program, Data, and Stack Segments



- A portion of memory is used as a stack with a processor register as a stack pointer

- PUSH: SP ← SP - 1
  M[SP] ← DR
- POP: DR ← M[SP]
  SP ← SP + 1

- Most computers do not provide hardware to check stack overflow (full stack) or underflow (empty stack) → must be done in software

REVERSE POLISH NOTATION

- **Arithmetic Expressions: A + B**

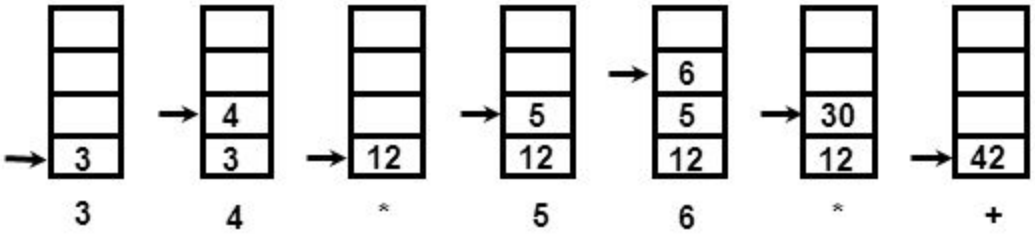
  A + B Infix notation
  + A B Prefix or Polish notation
  A B + Postfix or reverse Polish notation

  - The reverse Polish notation is very suitable for stack manipulation

- **Evaluation of Arithmetic Expressions**

  Any arithmetic expression can be expressed in parenthesis-free Polish notation, including reverse Polish notation

$$(3 * 4) + (5 * 6) \Rightarrow 3\ 4\ *\ 5\ 6\ *\ +$$



PROCESSOR ORGANIZATION

- In general, most processors are organized in one of 3 ways
  - Single register (Accumulator) organization
    - » Basic Computer is a good example
    - » Accumulator is the only general purpose register
  - General register organization
    - » Used by most modern computer processors
    - » Any of the registers can be used as the source or destination for computer operations
  - Stack organization
    - » All operations are done using the hardware stack
    - » For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack

INSTRUCTION FORMAT

- **Instruction Fields**

  OP-code field - specifies the operation to be performed
  Address field - designates memory address(es) or a processor register(s)
  Mode field - determines how the address field is to be interpreted (to get effective address or the operand)

- **The number of address fields in the instruction format depends on the internal organization of CPU**

- **The three most common CPU organizations:**

  Single accumulator organization:

```
ADD     X              /* AC ← AC + M[X]  */
```

  General register organization:

```
ADD     R1, R2, R3     /* R1 ← R2 + R3  */
ADD     R1, R2         /* R1 ← R1 + R2  */
MOV     R1, R2         /* R1 ← R2  */
ADD     R1, X          /* R1 ← R1 + M[X]  */
```

  Stack organization:

```
PUSH    X              /* TOS ← M[X]  */
ADD
```

THREE, AND TWO-ADDRESS INSTRUCTIONS

- **Three-Address Instructions**

Program to evaluate X = (A + B) * (C + D) :

```
ADD   R1, A, B      /* R1 ← M[A] + M[B]    */
ADD   R2, C, D      /* R2 ← M[C] + M[D]    */
MUL   X, R1, R2     /* M[X] ← R1 * R2      */
```

- Results in short programs (Advantage)
- Instruction becomes long (many bits)

- **Two-Address Instructions**

Program to evaluate X = (A + B) * (C + D) :

```
MOV   R1, A         /* R1 ← M[A]           */
ADD   R1, B         /* R1 ← R1 + M[A]      */
MOV   R2, C         /* R2 ← M[C]           */
ADD   R2, D         /* R2 ← R2 + M[D]      */
MUL   R1, R2        /* R1 ← R1 * R2        */
MOV   X, R1         /* M[X] ← R1           */
```

ONE, AND ZERO-ADDRESS INSTRUCTIONS

- **One-Address Instructions**
  - Use an implied AC register for all data manipulation
  - Program to evaluate X = (A + B) * (C + D) :

```
LOAD    A    /* AC ← M[A]          */
ADD     B    /* AC ← AC + M[B]     */
STORE   T    /* M[T] ← AC          */
LOAD    C    /* AC ← M[C]          */
ADD     D    /* AC ← AC + M[D]     */
MUL     T    /* AC ← AC * M[T]     */
STORE   X    /* M[X] ← AC          */
```

- **Zero-Address Instructions**
  - Can be found in a stack-organized computer
  - Program to evaluate X = (A + B) * (C + D) :

```
PUSH    A    /* TOS ← A            */
PUSH    B    /* TOS ← B            */
ADD          /* TOS ← (A + B)      */
PUSH    C    /* TOS ← C            */
PUSH    D    /* TOS ← D            */
ADD          /* TOS ← (C + D)      */
MUL          /* TOS ← (C + D) * (A + B)  */
POP     X    /* M[X] ← TOS         */
```

ADDRESSING MODES

- Addressing Modes

  * Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)

  * Variety of addressing modes

    - to give programming flexibility to the user
    - to use the bits in the address field of the instruction efficiently

TYPES OF ADDRESSING MODES

- **Implied Mode**

  Address of the operands are specified implicitly in the definition of the instruction
  - No need to specify address in the instruction
  - EA = AC, or EA = Stack[SP]
  - Examples from Basic Computer
    CLA, CME, INP

- **Immediate Mode**

  Instead of specifying the address of the operand, operand itself is specified
  - No need to specify address in the instruction
  - However, operand itself needs to be specified
  - Sometimes, require more bits than the address
  - Fast to acquire an operand

TYPES OF ADDRESSING MODES

- **Register Mode**

  Address specified in the instruction is the register address
  - Designated operand need to be in a register
  - Shorter address than the memory address
  - Saving address field in the instruction
  - Faster to acquire an operand than the memory addressing
  - EA = IR(R) (IR(R): Register field of IR)

- **Register Indirect Mode**

  Instruction specifies a register which contains the memory address of the operand
  - Saving instruction bits since register address is shorter than the memory address
  - Slower to acquire an operand than both the register addressing or memory addressing
  - EA = [IR(R)] ([x]: Content of x)

- **Autoincrement or Autodecrement Mode**
  - When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically

TYPES OF ADDRESSING MODES

- **Direct Address Mode**

  Instruction specifies the memory address which can be used directly to access the memory

  - Faster than the other memory addressing modes
  - Too many bits are needed to specify the address for a large physical memory space
  - EA = IR(addr) (IR(addr): address field of IR)

- **Indirect Addressing Mode**

  The address field of an instruction specifies the address of a memory location that contains the address of the operand

  - When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits
  - Slow to acquire an operand because of an additional memory access
  - EA = M[IR(address)]

TYPES OF ADDRESSING MODES

- **Relative Addressing Modes**

  The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand

  - Address field of the instruction is short
  - Large physical memory can be accessed with a small number of address bits
  - EA = f(IR(address), R), R is sometimes implied

3 different Relative Addressing Modes depending on R;

* **PC Relative Addressing Mode** (R = PC)
  - EA = PC + IR(address)
* **Indexed Addressing Mode** (R = IX, where IX: Index Register)
  - EA = IX + IR(address)
* **Base Register Addressing Mode** (R = BAR, where BAR: Base Address Register)
  - EA = BAR + IR(address)

ADDRESSING MODES - EXAMPLES -

PC = 200

R1 = 400

XR = 100

AC

Address	Memory	
200	Load to AC	Mode
201	Address = 500	
202	Next instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

Addressing Mode	Effective Address			Content of AC
Direct address	500	/* AC ← (500)	*/	800
Immediate operand	-	/* AC ← 500	*/	500
Indirect address	800	/* AC ← ((500))	*/	300
Relative address	702	/* AC ← (PC+500)	*/	325
Indexed address	600	/* AC ← (RX+500)	*/	900
Register	-	/* AC ← R1	*/	400
Register indirect	400	/* AC ← (R1)	*/	700
Autoincrement	400	/* AC ← (R1)+	*/	700
Autodecrement	399	/* AC ← -(R)	*/	450

DATA TRANSFER INSTRUCTIONS

- Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

- Data Transfer Instructions with Different Addressing Modes

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ADR	AC ← M[PC + ADR]
Immediate operand	LD #NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[ADR + XR]
Register	LD R1	AC ← R1
Register indirect	LD (R1)	AC ← M[R1]
Autoincrement	LD (R1)+	AC ← M[R1], R1 ← R1 + 1
Autodecrement	LD -(R1)	R1 ← R1 - 1, AC ← M[R1]

DATA MANIPULATION INSTRUCTIONS

- **Three Basic Types:** Arithmetic instructions
  Logical and bit manipulation instructions
  Shift instructions

- **Arithmetic Instructions**

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Borrow	SUBB
Negate(2's Complement)	NEG

- **Logical and Bit Manipulation Instructions**

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

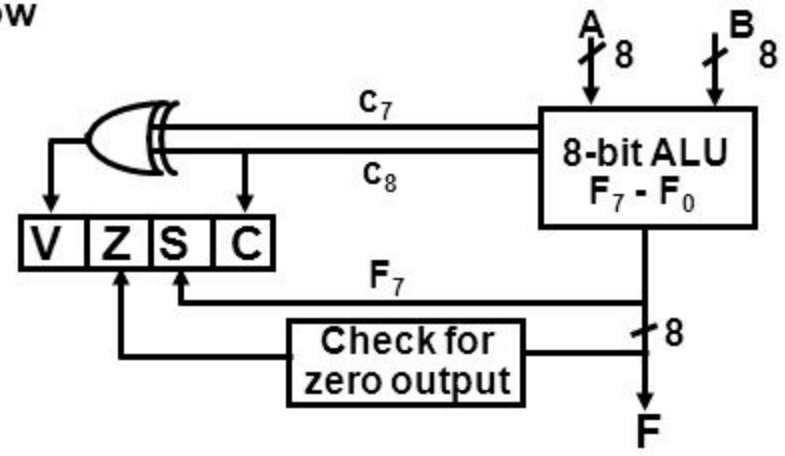
- **Shift Instructions**

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

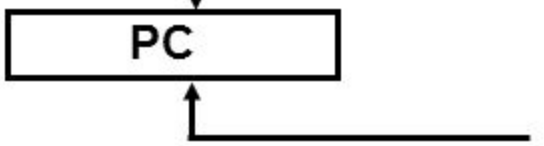
FLAG, PROCESSOR STATUS WORD

- In Basic Computer, the processor had several (status) flags – 1 bit value that indicated various information about the processor's state – E, FGI, FGO, I, IEN, R
- In some processors, flags like these are often combined into a register – the processor status register (PSR); sometimes called a processor status word (PSW)
- Common flags in PSW are
  - C (Carry): Set to 1 if the carry out of the ALU is 1
  - S (Sign): The MSB bit of the ALU's output
  - Z (Zero): Set to 1 if the ALU's output is all 0's
  - V (Overflow): Set to 1 if there is an overflow

Status Flag Circuit



PROGRAM CONTROL INSTRUCTIONS



+1
In-Line Sequencing (Next instruction is fetched from the next adjacent location in the memory)

Address from other source; Current Instruction, Stack, etc; Branch, Conditional Branch, Subroutine, etc

- Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RTN
Compare(by –)	CMP
Test(by AND)	TST

* CMP and TST instructions do not retain their results of operations (– and AND, respectively). They only set or clear certain Flags.

CONDITIONAL BRANCH INSTRUCTIONS

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned compare conditions (A - B)	
BHI	Branch if higher	A > B
BHE	Branch if higher or equal	A ≥ B
BLO	Branch if lower	A < B
BLOE	Branch if lower or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B
	Signed compare conditions (A - B)	
BGT	Branch if greater than	A > B
BGE	Branch if greater or equal	A ≥ B
BLT	Branch if less than	A < B
BLE	Branch if less or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B

SUBROUTINE CALL AND RETURN

- **Subroutine Call**
  Call subroutine
  Jump to subroutine
  Branch to subroutine
  Branch and save return address

- **Two Most Important Operations are Implied;**

  * Branch to the beginning of the Subroutine
    - Same as the Branch or Conditional Branch

  * Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine

- **Locations for storing Return Address**
  - Fixed Location in the subroutine (Memory)
  - Fixed Location in memory
  - In a processor Register
  - In memory *stack*
    - most efficient way

```
CALL
  SP ← SP - 1
  M[SP] ← PC
  PC ← EA

RTN
  PC ← M[SP]
  SP ← SP + 1
```

PROGRAM INTERRUPT

Types of Interrupts

External interrupts

External Interrupts initiated from the outside of CPU and Memory
- I/O Device → Data transfer request or Data transfer complete
- Timing Device → Timeout
- Power Failure
- Operator

Internal interrupts (traps)

Internal Interrupts are caused by the currently running program
- Register, Stack Overflow
- Divide by zero
- OP-code Violation
- Protection Violation

Software Interrupts

Both External and Internal Interrupts are initiated by the computer HW.
Software Interrupts are initiated by the executing an instruction.
- Supervisor Call → Switching from a user mode to the supervisor mode
  → Allows to execute a certain class of operations which are not allowed in the user mode

INTERRUPT PROCEDURE

Interrupt Procedure and Subroutine Call

- The interrupt is usually initiated by an internal or an external signal rather than from the execution of an instruction (except for the software interrupt)

- The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction

- An interrupt procedure usually stores all the information necessary to define the state of CPU rather than storing only the PC.

The state of the CPU is determined from;
- Content of the PC
- Content of all processor registers
- Content of status bits

Many ways of saving the CPU state depending on the CPU architectures

COMPLEX INSTRUCTION SET COMPUTER

- Continuing growth in semiconductor memory and microprogramming
  - ⇒ A much richer and complicated instruction sets and addressing modes
  - ⇒ Complex Instruction Set Computers (CISC)
- Richer instruction sets would simplify compilers
- Richer instruction sets would move as much functions to the hardware as possible
- Richer instruction sets would improve *architecture quality*
- One goal for CISC machines was to have a machine language instruction to match each high-level language statement type

VARIABLE LENGTH INSTRUCTIONS

- The large number of instructions means a greater number of bits to specify them
- The large number of instructions and addressing modes led CISC machines to have variable length instruction formats
- In order to manage this large number of opcodes efficiently, they were encoded with different lengths:
  - More frequently used instructions were encoded using short opcodes.
  - Less frequently used ones were assigned longer opcodes.
- Also, multiple operand instructions could specify different addressing modes for each operand
  - For example,
    - » Operand 1 could be a directly addressed register,
    - » Operand 2 could be an indirectly addressed memory location,
    - » Operand 3 (the destination) could be an indirectly addressed register.
- All of this led to the need to have different length instructions in different situations, depending on the opcode and operands used

VARIABLE LENGTH INSTRUCTIONS

- For example, an instruction that only specifies register operands may only be two bytes in length
  - One byte to specify the instruction and addressing mode
  - One byte to specify the source and destination registers.
- An instruction that specifies memory addresses for operands may need five bytes
  - One byte to specify the instruction and addressing mode
  - Two bytes to specify each memory address
    - » Maybe more if there's a large amount of memory.
- Variable length instructions greatly complicate the fetch and decode problem for a processor
- The circuitry to recognize the various instructions and to properly fetch the required number of bytes for operands is very complex

COMPLEX INSTRUCTION SET COMPUTER

- Another characteristic of CISC computers is that they have instructions that act directly on memory addresses
  - For example,

    ADD L1, L2, L3

    that takes the contents of *M*[*L1*] adds it to the contents of *M*[*L2*] and stores the result in location *M*[*L3*]
- An instruction like this takes three memory access cycles to execute
- The problems with CISC computers are
  - The complexity of the design may slow down the processor,
  - The complexity of the design may result in costly errors in the processor design and implementation,
  - Many of the instructions and addressing modes are used rarely, if ever

SUMMARY: CRITICISMS ON CISC

High Performance General Purpose Instructions

- Complex Instruction
  - → Format, Length, Addressing Modes
  - → Complicated instruction cycle control due to the complex decoding HW and decoding process

- Multiple memory cycle instructions
  - → Operations on memory data
  - → Multiple memory accesses/instruction

- Microprogrammed control is necessity
  - → Microprogram control storage takes substantial portion of CPU chip area
  - → Semantic Gap is large between machine instruction and microinstruction

- General purpose instruction set includes all the features required by individually different applications
  - → When any one application is running, all the features required by the other applications are extra burden to the application

REDUCED INSTRUCTION SET COMPUTERS

- In the late ‘70s and early ‘80s there was a reaction to the shortcomings of the CISC style of processors
- Reduced Instruction Set Computers (RISC) were proposed as an alternative
- The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time

- RISC processors often feature:
  - Few instructions
  - Few addressing modes
  - Only load and store instructions access memory
  - All other operations are done using on-processor registers
  - Fixed length instructions
  - Single cycle execution of instructions
  - The control unit is hardwired, not microprogrammed

REDUCED INSTRUCTION SET COMPUTERS

- Since all but the load and store instructions use only registers for operands, only a few addressing modes are needed
- By having all instructions the same length, reading them in is easy and fast
- The fetch and decode stages are simple, looking much more like Mano's Basic Computer than a CISC machine
- The instruction and address formats are designed to be easy to decode
- Unlike the variable length CISC instructions, the opcode and register fields of RISC instructions can be decoded simultaneously
- The control logic of a RISC processor is designed to be simple and fast
- The control logic is simple because of the small number of instructions and the simple addressing modes
- The control logic is hardwired, rather than microprogrammed, because hardwired control is faster

ARCHITECTURAL METRIC

$$A \leftarrow B + C$$
$$B \leftarrow A + C$$
$$D \leftarrow D - B$$

- Register-to-register (Reuse of operands)

8	4	16
Load	rB	B
Load	rC	C
Add	rA	rB rC
Store	rA	A
Add	rB	rA rC
Store	rB	B
Load	rD	D
Sub	rD	rD rB
Store	rD	D

I = 228b
D = 192b
M = 420b

- Register-to-register (Compiler allocates operands in registers)

8	4	4	4
Add	rA	rB	rC
Add	rB	rA	rC
Sub	rD	rD	rB

I = 60b
D = 0b
M = 60b

- Memory-to-memory

8	16	16	16
Add	B	C	A
Add	A	C	B
Sub	B	D	D

I = 168b
D = 288b
M = 456b

REGISTERS

- By simplifying the instructions and addressing modes, there is space available on the chip or board of a RISC CPU for more circuits than with a CISC processor
- This extra capacity is used to
  - Pipeline instruction execution to speed up instruction execution
  - Add a large number of registers to the CPU

PIPELINING

- A very important feature of many RISC processors is the ability to execute an instruction each clock cycle
- This may seem nonsensical, since it takes at least once clock cycle each to fetch, decode and execute an instruction.
- It is however possible, because of a technique known as pipelining
  - Study later
- Pipelining is the use of the processor to work on different phases of multiple instructions in parallel

PIPELINING

- For instance, at one time, a pipelined processor may be
  - Executing instruction i_t
  - Decoding instruction i_{t+1}
  - Fetching instruction i_{t+2} from memory
- So, if we're running three instructions at once, and it takes an average instruction three cycles to run, the CPU is executing an average of an instruction a clock cycle
- As we'll see when we cover it in depth, there are complications
  - For example, what happens to the pipeline when the processor branches
- However, pipelined execution is an integral part of all modern processors, and plays an important role

REGISTERS

- By having a large number of general purpose registers, a processor can minimize the number of times it needs to access memory to load or store a value
- This results in a significant speed up, since memory accesses are *much* slower than register accesses
- Register accesses are fast, since they just use the bus on the CPU itself, and any transfer can be done in one clock cycle
- To go off-processor to memory requires using the much slower memory (or system) bus
- It may take many clock cycles to read or write to memory across the memory bus
  - The memory bus hardware is usually slower than the processor
  - There may even be competition for access to the memory bus by other devices in the computer (*e.g.* disk drives)
- So, for this reason alone, a RISC processor may have an advantage over a comparable CISC processor, since it only needs to access memory
  - for its instructions, and
  - occasionally to load or store a memory value

REGISTER WINDOW APPROACH

- **Observations**

  - Frequency of HLL Operations

    ⇒ Procedure call/return is the most time consuming operations

  - Locality of Procedure Nesting

    ⇒ The depth of procedure activation fluctuates within a relatively narrow range

  - A typical procedure employs only a few passed parameters and local variables

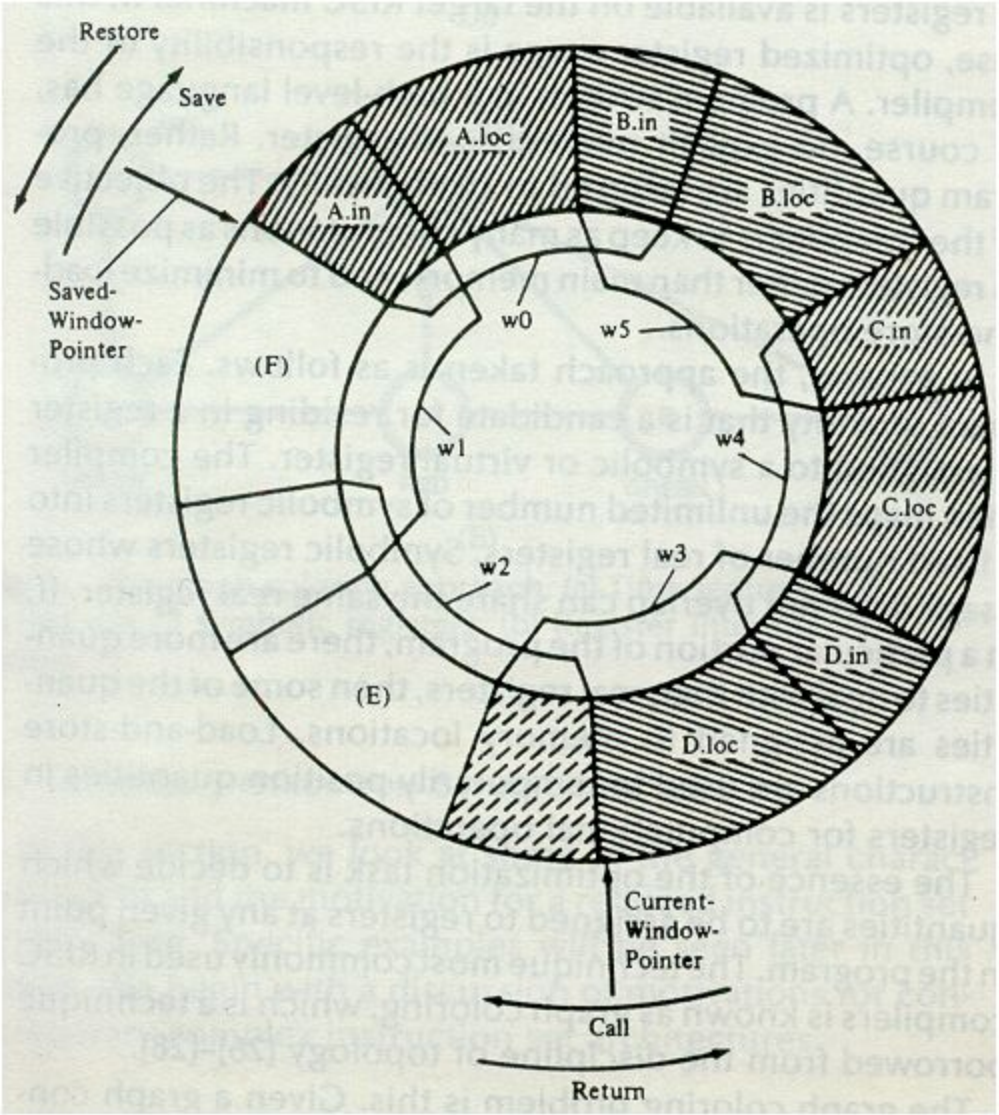
- **Solution**

  - Use multiple small sets of registers (windows), each assigned to a different procedure

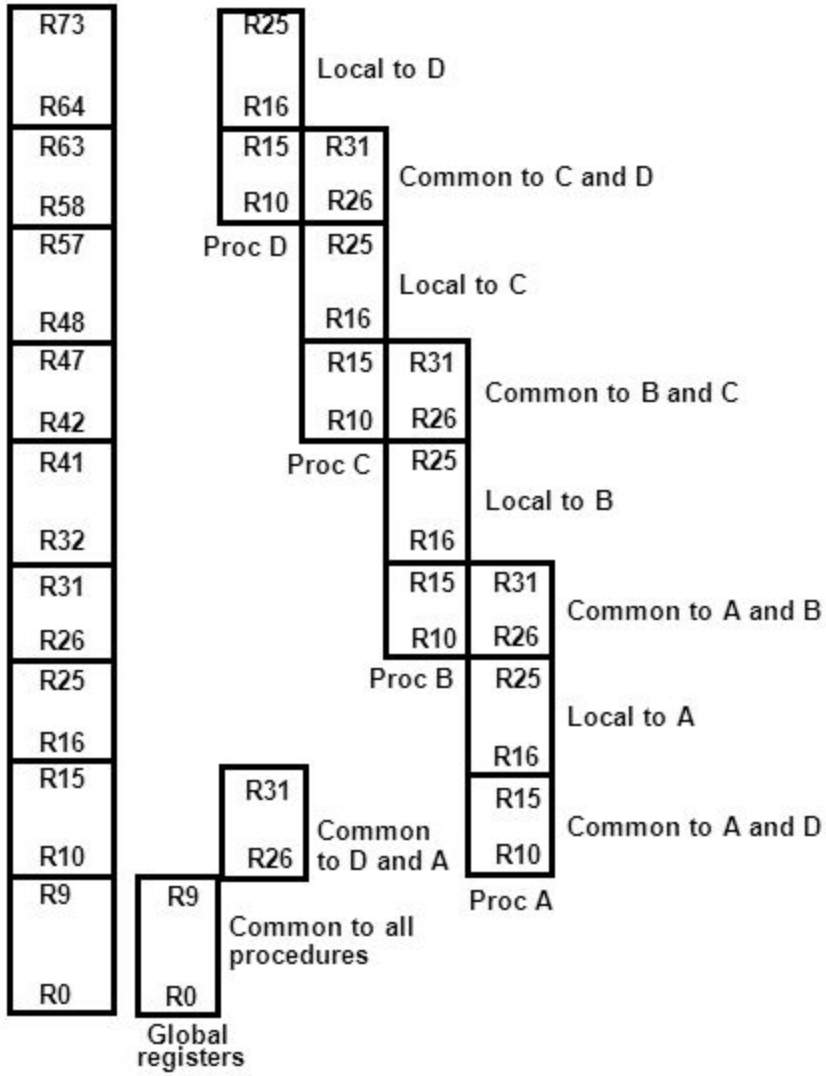
  - A procedure call automatically switches the CPU to use a different window of registers, rather than saving registers in memory

  - Windows for adjacent procedures are overlapped to allow parameter passing

CIRCULAR OVERLAPPED REGISTER WINDOWS



OVERLAPPED REGISTER WINDOWS



OVERLAPPED REGISTER WINDOWS

- There are three classes of registers:
  - Global Registers
    - » Available to all functions
  - Window local registers
    - » Variables local to the function
  - Window shared registers
    - » Permit data to be shared without actually needing to copy it
- Only one register window is active at a time
  - The active register window is indicated by a pointer
- When a function is called, a new register window is activated
  - This is done by incrementing the pointer
- When a function calls a new function, the high numbered registers of the calling function window are shared with the called function as the low numbered registers in its register window
- This way the caller's high and the called function's low registers overlap and can be used to pass parameters and results

OVERLAPPED REGISTER WINDOWS

- In addition to the overlapped register windows, the processor has some number of registers, *G*, that are global registers
  - This is, all functions can access the global registers.
- The advantage of overlapped register windows is that the processor does not have to push registers on a stack to save values and to pass parameters when there is a function call
  - Conversely, pop the stack on a function return
- This saves
  - Accesses to memory to access the stack.
  - The cost of copying the register contents at all
- And, since function calls and returns are so common, this results in a significant savings relative to a stack-based approach

CHARACTERISTICS OF RISC

- RISC Characteristics
  - Relatively few instructions
  - Relatively few addressing modes
  - Memory access limited to load and store instructions
  - All operations done within the registers of the CPU
  - Fixed-length, easily decoded instruction format
  - Single-cycle instruction format
  - Hardwired rather than microprogrammed control

- Advantages of RISC
  - VLSI Realization
  - Computing Speed
  - Design Costs and Reliability
  - High Level Language Support

ADVANTAGES OF RISC

- **VLSI Realization**

  Control area is considerably reduced

  Example:
  RISC I: 6%
  RISC II: 10%
  MC68020: 68%
  general CISCs: ~50%

  ⇒ RISC chips allow a large number of registers on the chip

  - Enhancement of performance and HLL support
  - Higher regularization factor and lower VLSI design cost

  The GaAs VLSI chip realization is possible

- **Computing Speed**
  - Simpler, smaller control unit ⇒ faster
  - Simpler instruction set; addressing modes; instruction format
    ⇒ faster decoding
  - Register operation ⇒ faster than memory operation
  - Register window ⇒ enhances the overall speed of execution
  - Identical instruction length, One cycle instruction execution
    ⇒ suitable for pipelining ⇒ faster

PIPELINING AND VECTOR PROCESSING

- Parallel Processing
- Pipelining
- Arithmetic Pipeline
- Instruction Pipeline
- RISC Pipeline
- Vector Processing
- Array Processors

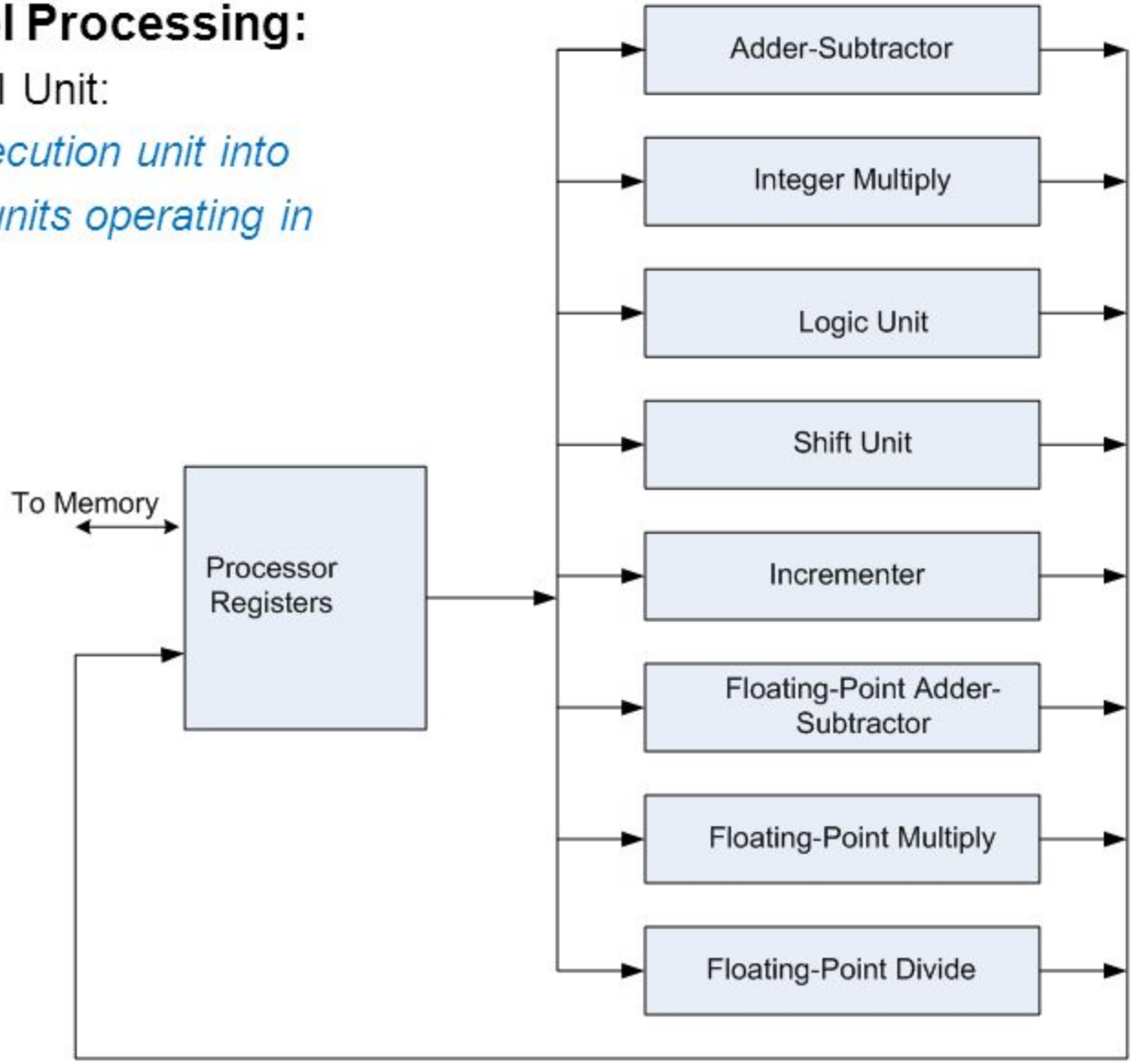
PARALLEL PROCESSING

- Parallel processing is a term used for a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system.

PARALLEL PROCESSING

- **Example of parallel Processing:**
  - Multiple Functional Unit:

    Separate the execution unit into eight functional units operating in parallel.



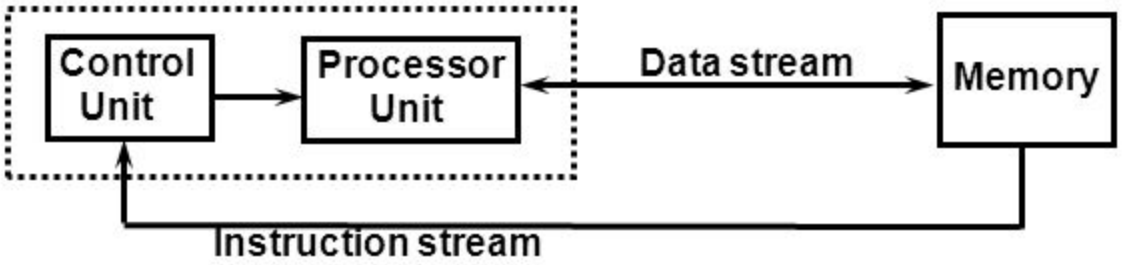
PARALLEL COMPUTERS

Architectural Classification

- **Flynn's classification**
  - » Based on the multiplicity of *Instruction Streams* and *Data Streams*
  - » Instruction Stream
    - Sequence of Instructions read from memory
  - » Data Stream
    - Operations performed on the data in the processor

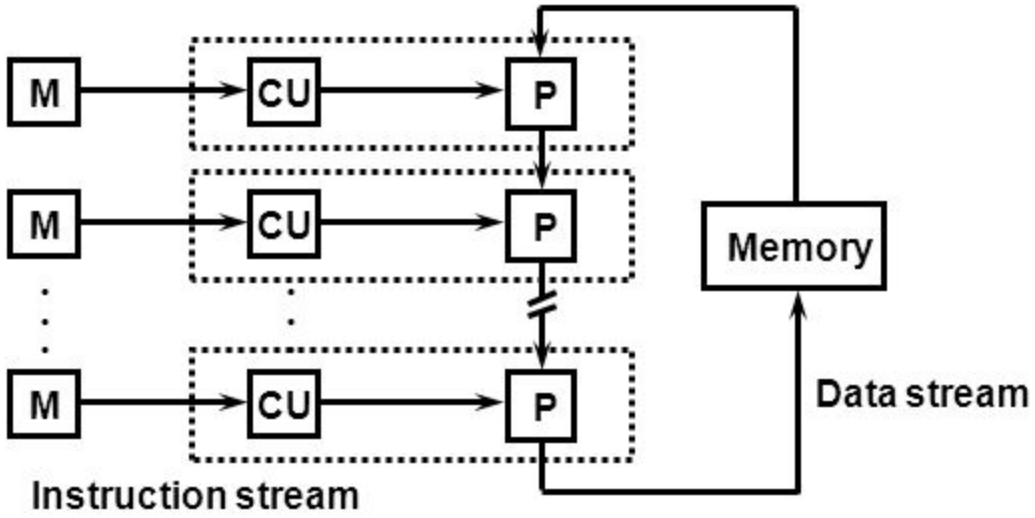
		Number of *Data Streams*	
		Single	Multiple
Number of *Instruction Streams*	Single	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

SISD COMPUTER SYSTEMS



- **Characteristics:**
  - One control unit, one processor unit, and one memory unit
  - Parallel processing may be achieved by means of:
    - ✓multiple functional units
    - ✓pipeline processing

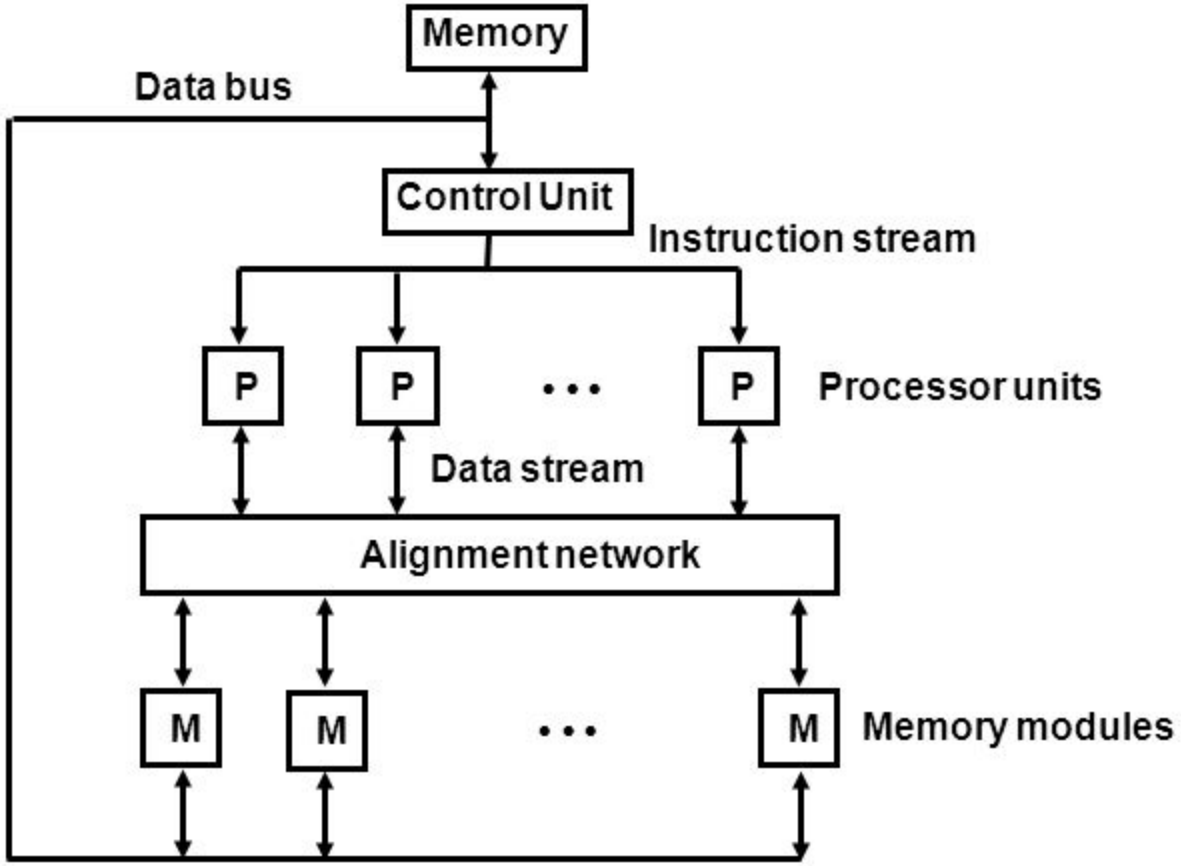
MISD COMPUTER SYSTEMS



Characteristics

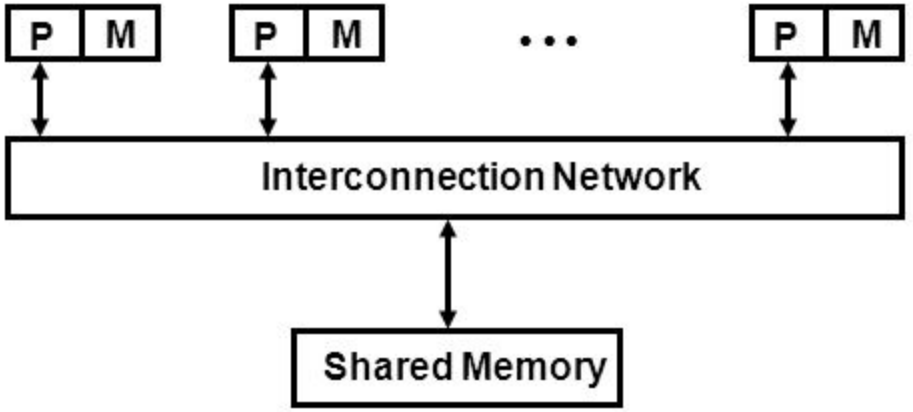
- There is no computer at present that can be classified as MISD

SIMD COMPUTER SYSTEMS



- **Characteristics**
  - Only one copy of the program exists
  - A single controller executes one instruction at a time

MIMD COMPUTER SYSTEMS



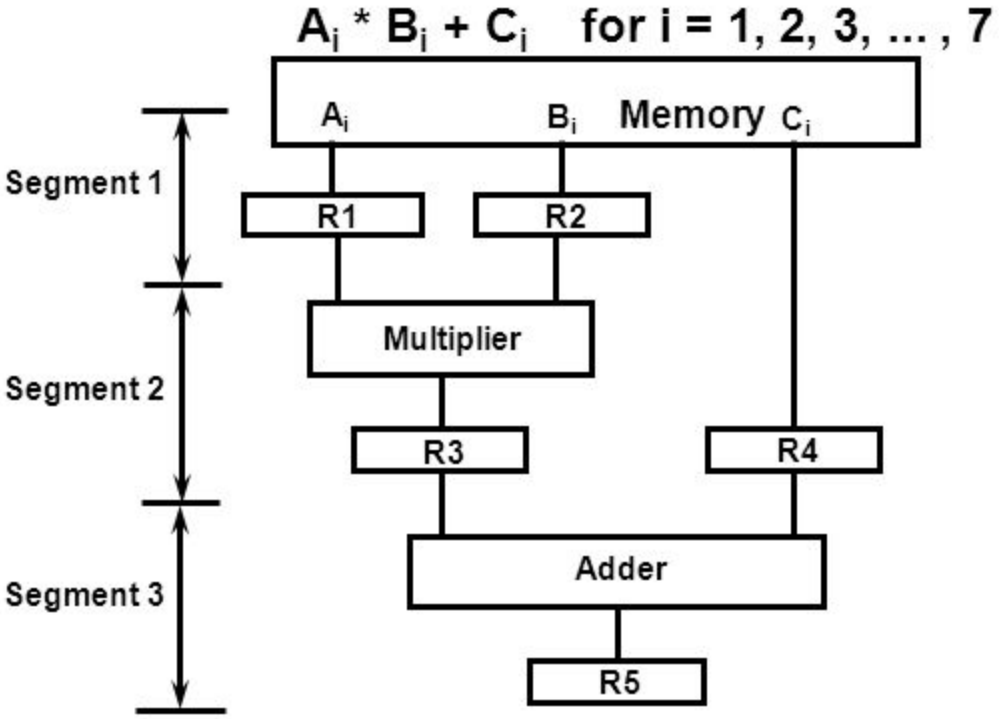
- **Characteristics:**
  - Multiple processing units (multiprocessor system)
  - Execution of multiple instructions on multiple data

- **Types of MIMD computer systems**
  - Shared memory multiprocessors
  - Message-passing multicomputers (multicomputer system)

- The main difference between multicomputer system and multiprocessor system is that the multiprocessor system is controlled by one operating system that provides interaction between processors and all the component of the system cooperate in the solution of a problem.

PIPELINING

- A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.



Suboperations in each segment:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$ Load A_i and B_i

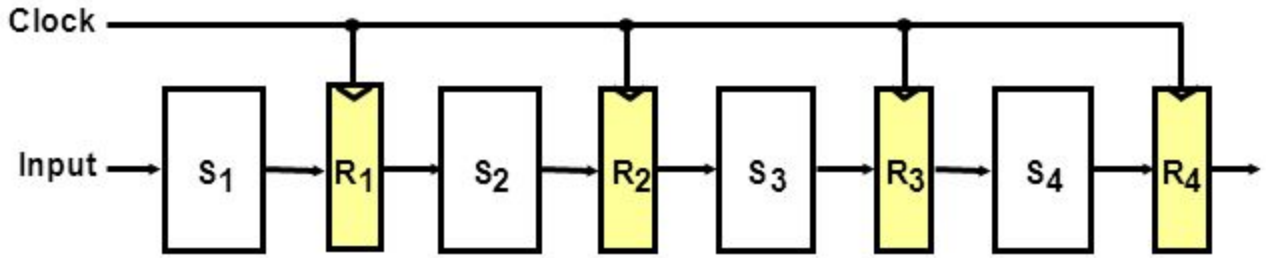
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$ Multiply

OPERATIONS IN EACH PIPELINE STAGE

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1	---	---	-------
2	A2	B2	A1 * B1	C1	-------
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

GENERAL PIPELINE

- **General Structure of a 4-Segment Pipeline**



- **Space-Time Diagram**

  The following diagram shows 6 tasks T1 through T6 executed in 4 segments.

Clock cycles

Segment	1	2	3	4	5	6	7	8	9
1	T1	T2	T3	T4	T5	T6			
2		T1	T2	T3	T4	T5	T6		
3			T1	T2	T3	T4	T5	T6	
4				T1	T2	T3	T4	T5	T6

No matter how many segments, once the pipeline is full, it takes only one clock period to obtain an output.

PIPELINE SPEEDUP

Consider the case where a k-segment pipeline used to execute n tasks.

- n = 6 in previous example
- k = 4 in previous example

- **Pipelined Machine (k stages, n tasks)**
  - The first task t1 requires k clock cycles to complete its operation since there are k segments
  - The remaining n-1 tasks require n-1 clock cycles
  - The n tasks clock cycles = k+(n-1) (9 in previous example)
- **Conventional Machine (Non-Pipelined)**
  - Cycles to complete each task in nonpipeline = k
  - For n tasks, n cycles required is
- **Speedup (S)**
  - S = Nonpipeline time /Pipeline time
  - For n tasks: S = nk/(k+n-1)
  - As n becomes much larger than k-1; Therefore, S = nk/n = k

PIPELINE AND MULTIPLE FUNCTION UNITS

Example:

- 4-stage pipeline
- 100 tasks to be executed
- 1 task in non-pipelined system; 4 clock cycles

Pipelined System : $k + n - 1 = 4 + 99 = 103$ clock cycles

Non-Pipelined System : $n*k = 100 * 4 = 400$ clock cycles

Speedup : $S_k = 400 / 103 = 3.88$

Types of Pipelining

- Arithmetic Pipeline
- Instruction Pipeline

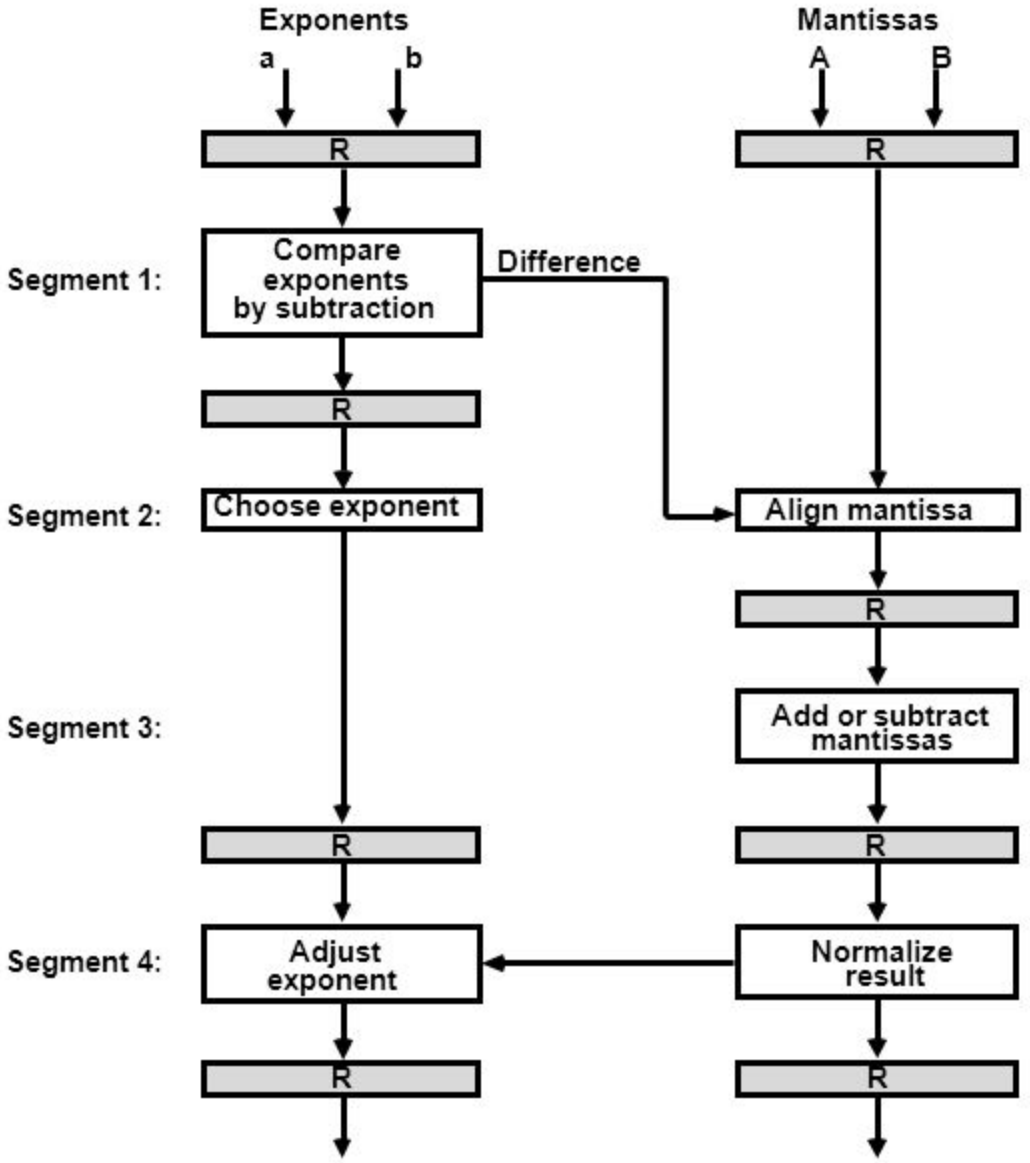
ARITHMETIC PIPELINE

Floating-point adder

[1] Compare the exponents
[2] Align the mantissa
[3] Add/sub the mantissa
[4] Normalize the result

$X = A \times 10^a = 0.9504 \times 10^3$
$Y = B \times 10^b = 0.8200 \times 10^2$

1) Compare exponents :
   $3 - 2 = 1$
2) Align mantissas
   $X = 0.9504 \times 10^3$
   $Y = 0.08200 \times 10^3$
3) Add mantissas
   $Z = 1.0324 \times 10^3$
4) Normalize result
   $Z = 0.10324 \times 10^4$



INSTRUCTION CYCLE

Pipeline processing can occur also in the instruction stream. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.

Six Phases* in an Instruction Cycle

[1] Fetch an instruction from memory
[2] Decode the instruction
[3] Calculate the effective address of the operand
[4] Fetch the operands from memory
[5] Execute the operation
[6] Store the result in the proper place

* Some instructions skip some phases
* Effective address calculation can be done in the part of the decoding phase
* Storage of the operation result into a register is done automatically in the execution phase

==> 4-Stage Pipeline

[1] FI: Fetch an instruction from memory
[2] DA: Decode the instruction and calculate the effective address of the operand
[3] FO: Fetch the operand
[4] EX: Execute the operation

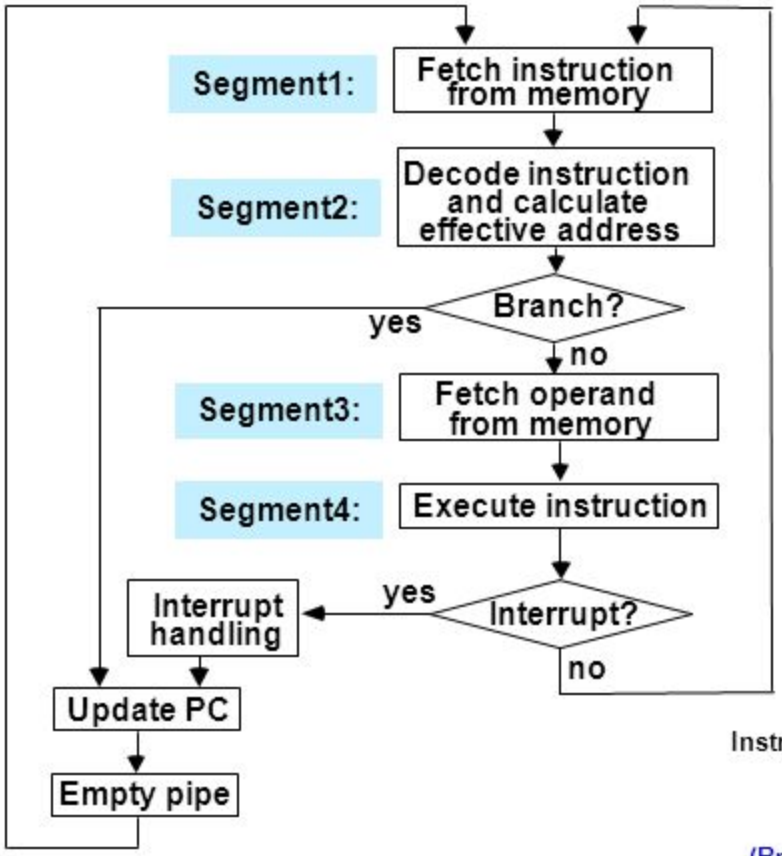
INSTRUCTION PIPELINE

Execution of Three Instructions in a 4-Stage Pipeline

Conventional

Pipelined

INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



Instruction (Branch)

Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Pipeline Conflicts

– **Pipeline Conflicts : 3 major difficulties**

1) **Resource conflicts:** memory access by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.

2) **Data dependency:** when an instruction depend on the result of a previous instruction, but this result is not yet available.

   Example: an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register.

3) **Branch difficulties:** branch and other instruction (interrupt, ret, ..) that change the value of PC.

RISC Computer

- **RISC (Reduced Instruction Set Computer)**
  - Machine with a very fast clock cycle that executes at the rate of one instruction per cycle.

- **Major Characteristic**
  1. Relatively few instructions
  2. Relatively few addressing modes
  3. Memory access limited to load and store instructions
  4. All operations done within the registers of the CPU
  5. Fixed-length, easily decoded instruction format
  6. Single-cycle instruction execution
  7. Hardwired rather than microprogrammed control
  8. Relatively large number of registers in the processor unit
  9. Efficient instruction pipeline
  10. Compiler support for efficient translation of high-level language programs into machine language programs

RISC PIPELINE

- **Instruction Cycle of Three-Stage Instruction Pipeline**

  I: Instruction Fetch

  A: Decode, Read Registers, ALU Operation

  E: Transfer the output of ALU to a register, memory, or PC.

- **Types of instructions**

  - Data Manipulation Instructions
  - Load and Store Instructions
  - Program Control Instructions

VECTOR PROCESSING

- There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems require a vast number of computations that will take a conventional computer days or even weeks to complete.

Vector Processing Applications

- Problems that can be efficiently formulated in terms of vectors and matrices
  - Long-range weather forecasting - Petroleum explorations
  - Seismic data analysis - Medical diagnosis
  - Aerodynamics and space flight simulations
  - Artificial intelligence and expert systems
  - Mapping the human genome
  - Image processing

Vector Processor (computer)

- Ability to process vectors, and matrices much faster than conventional computers

VECTOR PROGRAMMING

Fortran Language

```
      DO  20  I = 1, 100
20    C(I) = B(I) + A(I)
```

Conventional computer (Mahine language)

```
      Initialize I = 0
20    Read A(I)
      Read B(I)
      Store C(I) = A(I) + B(I)
      Increment I = I + 1
      If I ≤ 100 goto 20
```

Vector computer

```
C(1:100) = A(1:100) + B(1:100)
```

VECTOR PROGRAMMING

– Vector Instruction Format :

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length
ADD	A	B	C	100

– Matrix Multiplication

» 3 x 3 matrices multiplication :

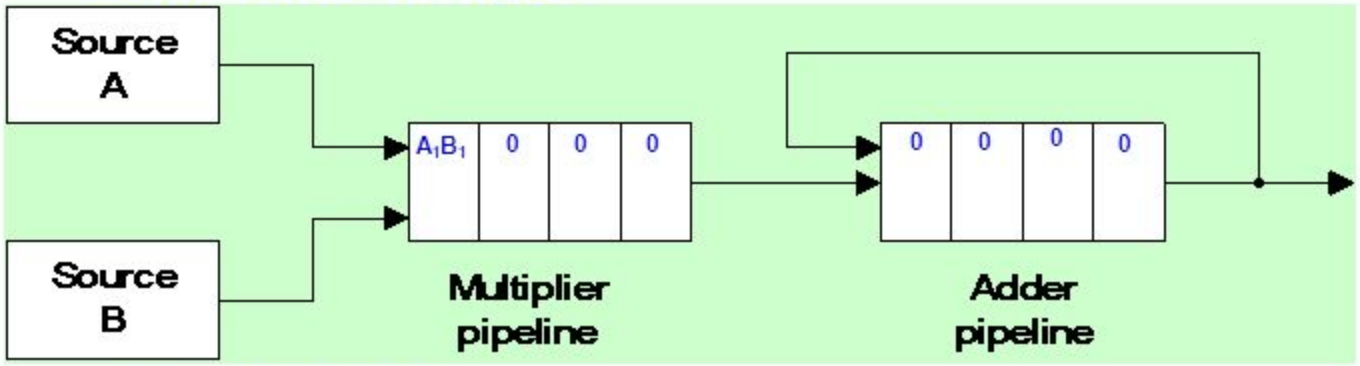
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

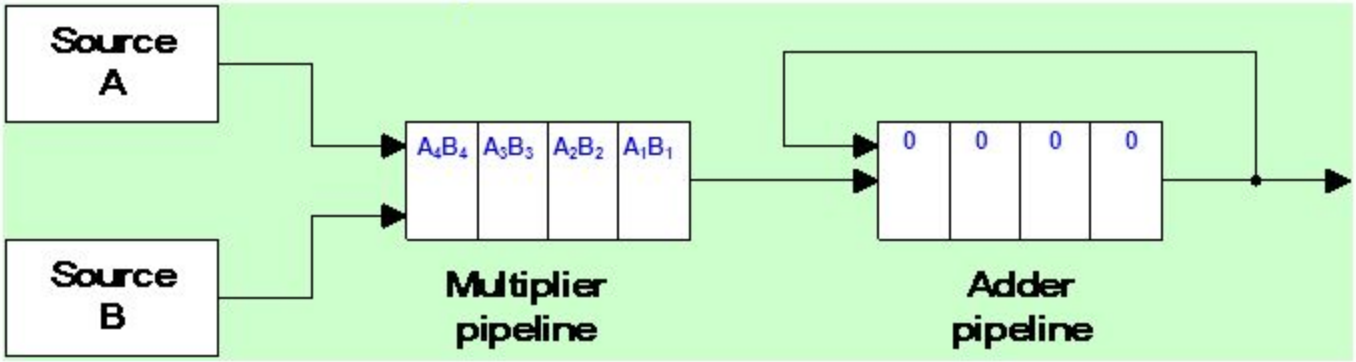
– Pipeline for calculating an inner product :
  » Floating point multiplier pipeline : 4 segments
  » Floating point adder pipeline : 4 segments

$$C = A_1B_1 + A_2B_2 + A_3B_3 + \cdots + A_kB_k$$

• after 1st clock input

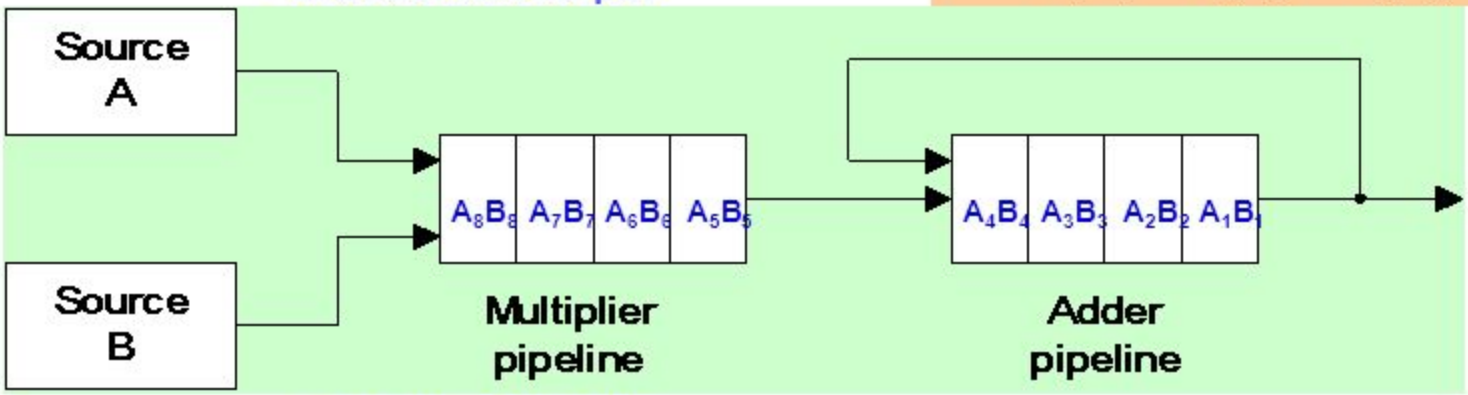


• after 4th clock input

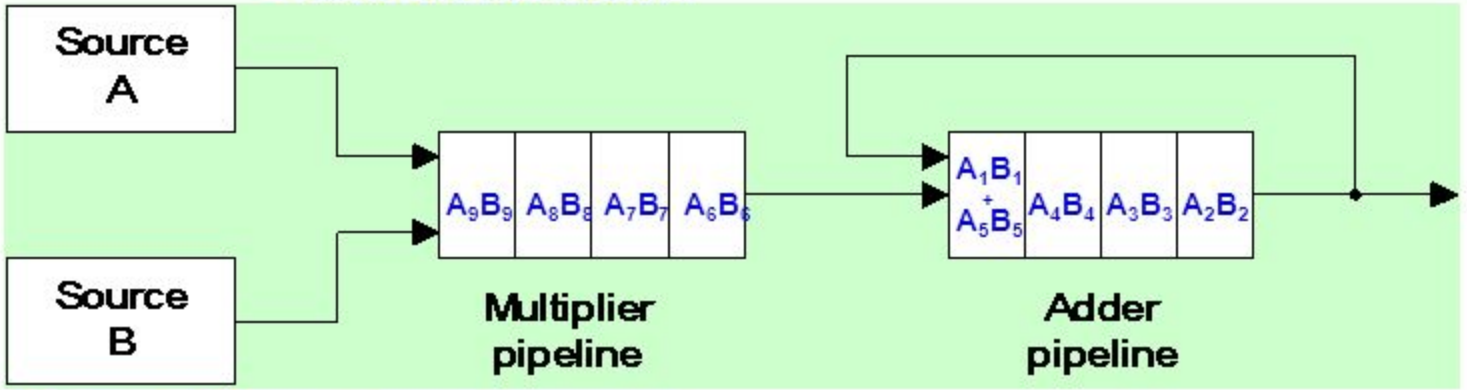


$$C = A_1B_1 + A_2B_2 + A_3B_3 + \cdots + A_kB_k$$

- after 8th clock input



- after 9th, 10th, 11th ,...



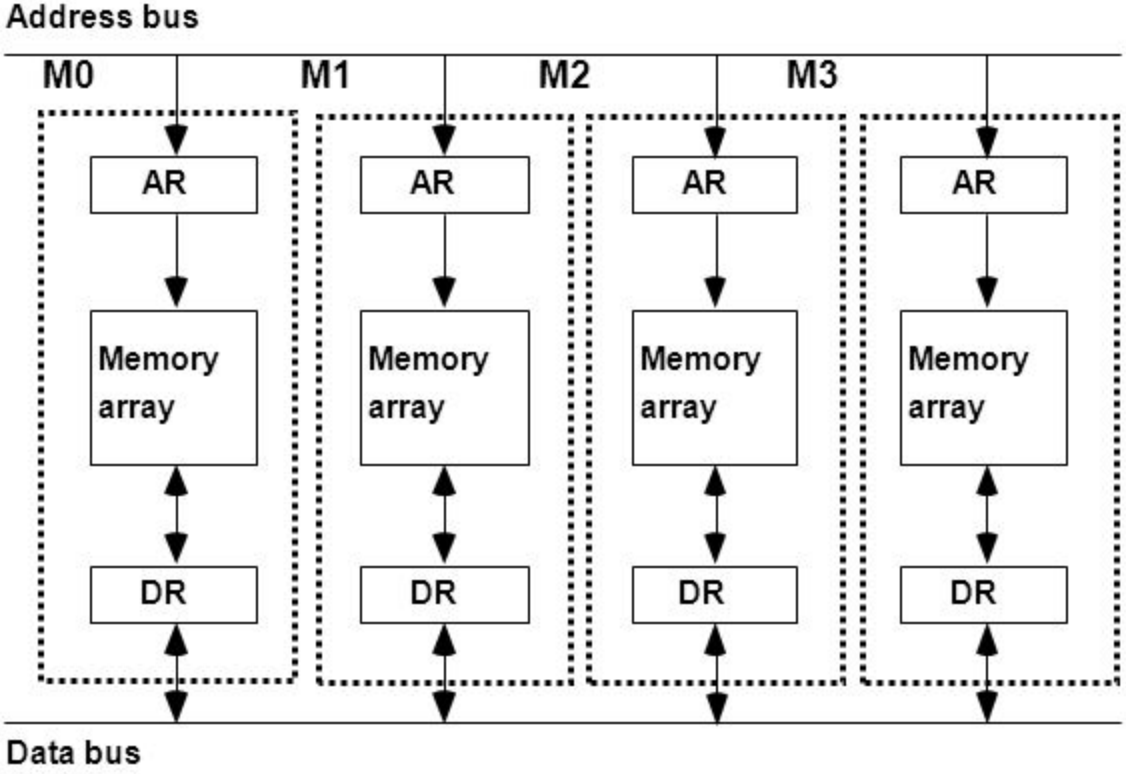
$$\begin{aligned} C = {} & A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \cdots \\ & + A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \cdots \\ & + A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \cdots \\ & + A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \cdots \end{aligned}$$

MEMORY INTERLEAVING

- Pipeline and vector processors often require simultaneous access to memory from tow or more sources.
- An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.
- An arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time.
- Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to common memory address and data buses.

- **Address Interleaving**
  - Different sets of addresses are assigned to different memory modules
  - For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other.

MEMORY INTERLEAVING



- A vector processor that uses an n-way interleaved memory can fetch n operands from n different modules. By staggering the memory access, the effective memory cycle time can be reduced by a factor close to the number of modules.
- A CPU with instruction pipeline can take advantage of multiple memory modules so that each segment in the pipeline can access memory independent of memory access from other segments.

Supercomputer

- Supercomputer = Vector Instruction + Pipelined floating-point arithmetic
- High computational speed, fast and large memory system.
- Extensive use of parallel processing.
- It is equipped with multiple functional units and each unit has its own pipeline configuration.
- Optimized for the type of numerical calculations involving vectors and matrices of floating-point numbers.
- Limited in their use to a number of scientific applications:
  - *numerical weather forecasting,*
  - *seismic wave analysis,*
  - *space research.*
- They have limited use and limited market because of their high price.

Supercomputer

- Performance Evaluation Index
  - » MIPS : Million Instruction Per Second
  - » FLOPS : Floating-point Operation Per Second
    - megaflops : 10^6, gigaflops : 10^9

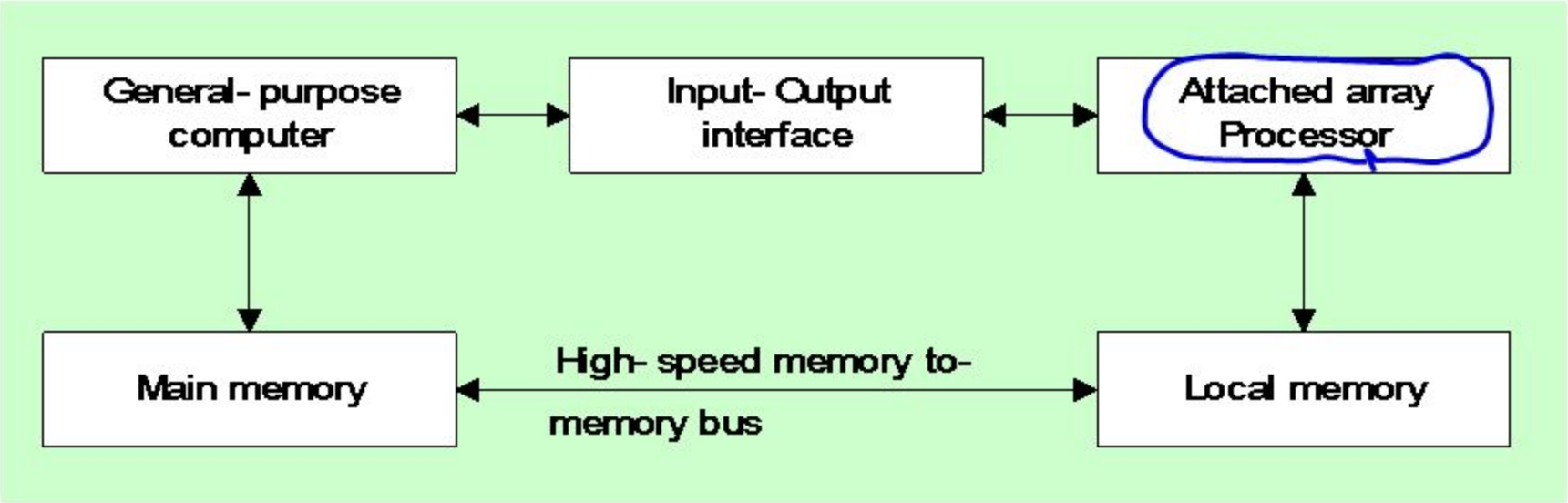
- Cray supercomputer :
  - » Cray-1 : 80 megaflops, (1976)
  - » Cray-2 : 12 times more powerful than the Cray-1
- VP supercomputer : Fujitsu
  - » VP-200 : 300 megaflops, 83 vector instruction, 195 scalar instruction
  - » VP-2600 : 5 gigaflops

9-7 Array Processors

- Performs computations on large arrays of data
  - » **Attached array processor :**
    - Auxiliary processor attached to a general purpose computer to improve the numerical computation performance.
  - » **SIMD array processor :**
    - Computer with multiple processing units operating in parallel
      - Vector $C = A + B$ $\quad c_i = a_i + b_i$
- Although both types manipulate vectors, their internal organization is different.

9-7 Array Processors

Attached array processor



- Designed as a peripheral for complex scientific applications attached with a conventional host computer.
- The peripheral is treated like and external interface. The data are transferred from main memory to local memory through high-speed bus.
- The general-purpose computer without the attached processor serves the users that need conventional data processing.

Thank You