Computer Architecture
CSEN 3104
Lecture 1

Dr. Debranjan Sarkar

What is Computer Architecture ?

- Goal
  - Create a computing system
    - With best performance
    - Having least price
    - Requiring minimum energy consumption
    - That meets all the functional requirement
- Hardware components
  - Design
  - Select
  - Interconnection between different hardware devices
- Design Hardware – Software interface

Stored Program Computer (Von Neumann concept)

- Early computers were
  - Mostly not reprogrammable
  - Executed a single hardwired program
  - No program instructions -> no program storage
  - Some computers were programmable
    - But stored their programs on punched tapes
    - These were physically fed into the machine as and when needed.
- In late 1940s, John von Neumann gave the concept of storing instructions in computer memory
- This enabled a computer to perform a variety of tasks in sequence or intermittently

Von Neumann concept

- A program may be electronically stored in binary-number format in a memory device
- Now instructions may be modified by the computer
- A computer with a von Neumann architecture stores program and data in the same memory
- *Stored-program computer* is sometimes used as a synonym for von Neumann architecture, or IAS computer (as it was first developed at the Institute for Advanced Studies)
- The von Neumann architecture is also known as Princeton architecture

Von Neumann concept

- Stored program concept
- Both Data and instructions are stored in a single read-write memory
- Arithmetic and Logic Unit (ALU) is capable of operating on binary data
- The contents of the memory are addressable by location
- The computer executes the program in sequential fashion from one instruction to the next, unless explicitly modified.
- A program can modify itself when the computer executes the program

General structure of von Neumann Architecture

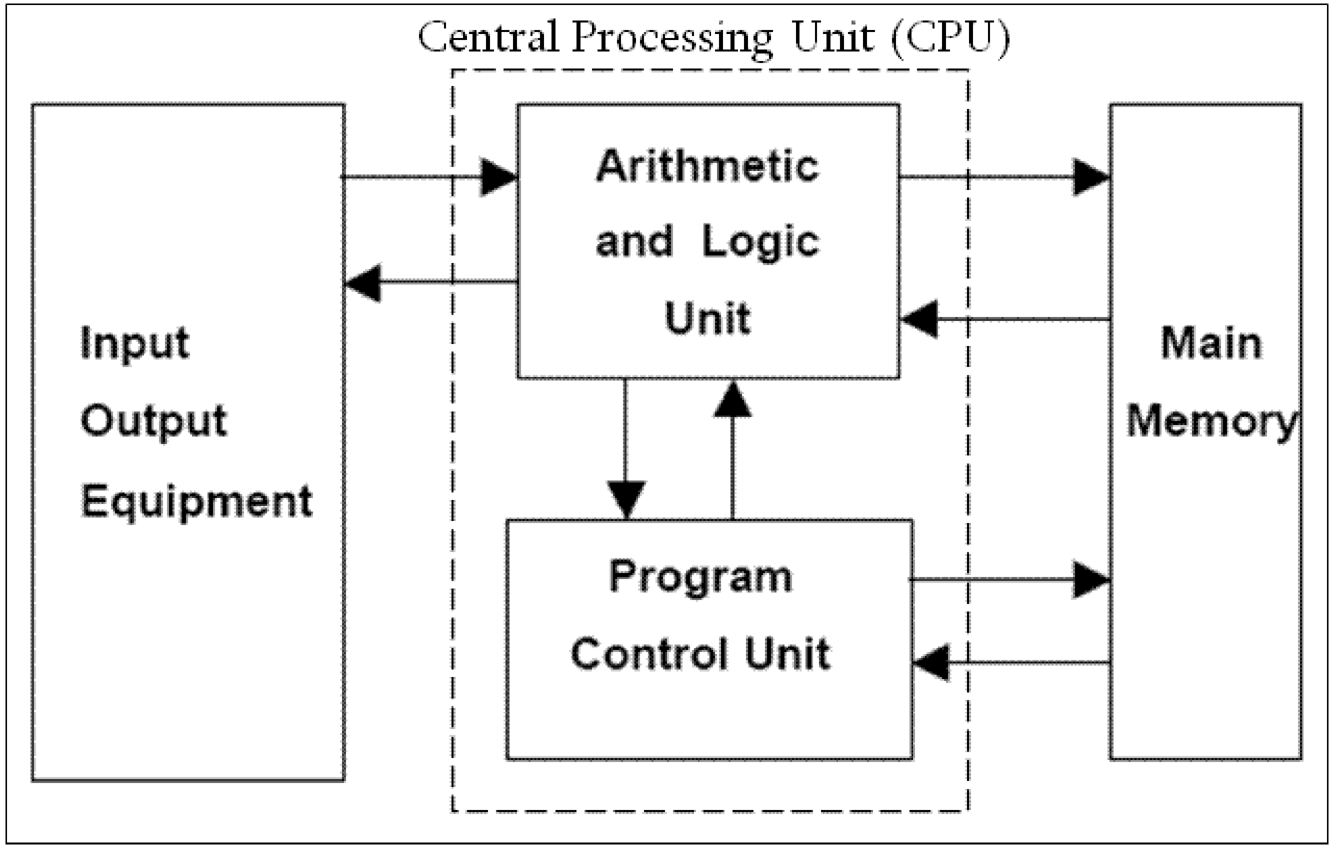


Figure : General structure of Von Neumann Architecture

What is von Neumann bottleneck?

- Von Neumann architecture requires memory access for instruction fetch and for data movement (from and to memory)
- Memory access is very slow compared to the speed of the CPU
- So CPU has to wait for a long to obtain instruction / data from memory
- This greatly degrades the performance of the Von Neumann computer
- Also, an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This often limits the performance of the system
- The degradation of performance due to CPU-memory speed disparity and due to sharing the same bus for instruction fetch and data read/write is referred to as Von Neumann bottleneck

How von Neumann bottleneck can be overcome?

- The performance is improved by using a special type of faster memory (called cache memory) between the CPU and the main memory. The access time of the cache memory is of the order of the speed of the CPU and hence there is almost no waiting time by the CPU for the required data

- By using separate instruction cache and data cache

- By moving some data into cache before it is requested (pre-fetching) to speed access in the event of a request

- By using RISC (Reduced Instruction Set Computer) architecture to limit access to main memory to a few load and store instructions. Other instructions have their operands in CPU registers (not in memory).

Difference between Von Neumann architecture and Harvard architecture

- The von Neumann architecture is a stored program concept
- It consists of a single memory for both program and data storage
- The system performance degrades as program and data cannot be fetched in one cycle
- Example: EDVAC (Electronic Discrete Variable Automatic Computer)

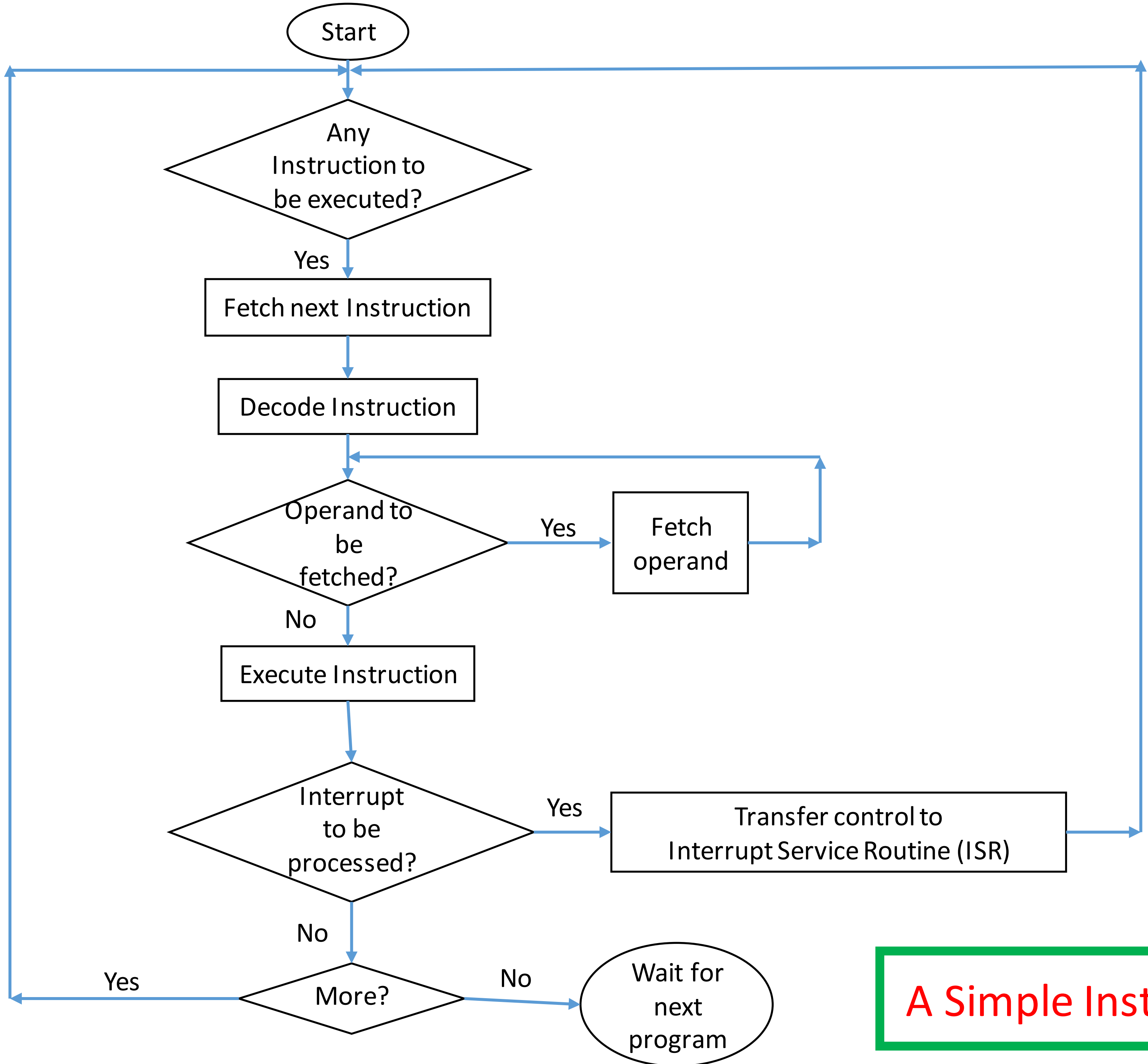
whereas

- Harvard architecture is also a stored program concept
- It has separate program and data memories
- Data memory and program memory can be of different width, type etc.
- Program and data can be fetched in one cycle – by using separate control signals: ‘program memory read’ and ‘data memory read’
- Example: Harvard Mark 1 computer

Thank you

Computer Architecture CSEN 3104 Lecture 2

Dr. Debranjan Sarkar



A Simple Instruction Cycle

Instruction Execution Mechanism

- A program is a set of instructions stored in memory
- The program is executed in the computer by going through a cycle for each instruction
- After the program is loaded onto the memory, the CPU fetches the first instruction
- Then the instruction is decoded to understand what actions the instruction dictates
- If required, it fetches the operand from the memory
- Then the CPU carries out those actions i.e. executes the instruction

Contd….

Instruction Execution Mechanism

- If no interrupt is pending to be serviced, the control is transferred to the next instruction
- In case some interrupt is pending to be serviced, the CPU transfers control to the Interrupt Service Routine (ISR)
- After execution of the ISR, control is transferred to the next instruction (from where it came to ISR)
- This cycle is repeated continuously by a computer's CPU, from boot up to shut down.
- The fetch–decode–execute cycle (also known as instruction cycle) is the basic operational process of a computer

Instruction Set Architecture

Operation Code	Mode	Address

- Instruction Format
- Operation Code
    - Example: Add, Sub, Complement etc.
- Address field
    - Memory location
    - Processor Register
    - Operand value
- Mode
    - Specifies the addressing mode to get the operand
    - Effective address of the operand
    - In some computer, no separate mode field and the addressing mode is specified in the instruction (opcode) itself
- Example: ADD R1, R0

Instruction Set Architecture

- In certain situations, special fields are used
  - Number of shifts in a SHIFT type instruction
  - Label field in a BRANCH type instruction
- Memory or Registers store the operand values on which the instructions are executed
- Memory addresses are used to specify operands stored in memory
- A register address (k-bit) specifies one out of 2^k registers in the CPU
- A CPU with 32 registers has a register address field of 5 bits

Thank you

Computer Architecture
CSEN 3104
Lecture 3

Dr. Debranjan Sarkar

Various types of Instruction Set Architecture

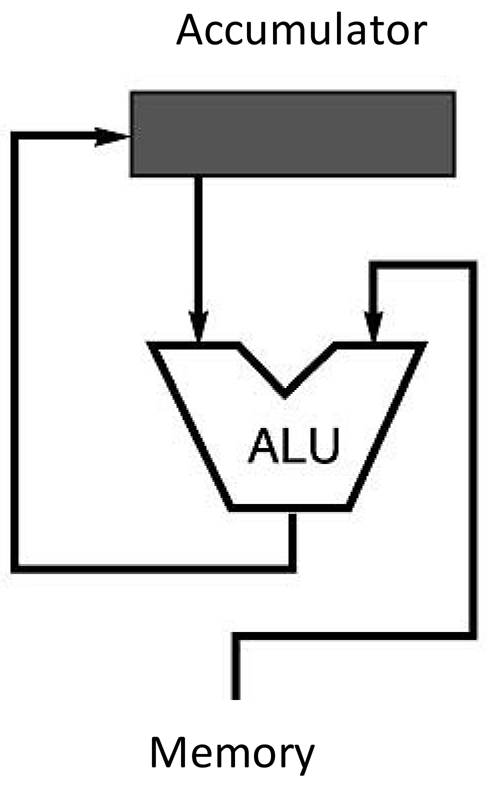
- Accumulator architecture
- General Register based architecture
  - Register-Memory architecture
  - Memory-Memory architecture
- Register (Load/ store) architecture
- Stack architecture

Accumulator architecture

- A single register, called the Accumulator is used to
  - process all the instructions
  - store the operand before the operation
  - store intermediate results
  - store the result after the operation
- Instruction format has only one operand (in register or memory)
- Accumulator almost always implicitly used
- This type of CPU is known as one-address machine
- Example: MULT X [X = address of the operand]
- (AC) $\leftarrow$ (AC) * mem[X]

Accumulator architecture

- Example: A*B - (X+Y*Z)
  - load Z
  - mul Y
  - add X
  - store C
  - load A
  - mul B
  - sub C
- Advantages
  - Very low hardware requirements
  - Easy to design and understand
  - Short instruction and less memory space
  - Instruction cycle is faster
- Disadvantages
  - Accumulator becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Program size increases as many short instructions are required
  - High memory traffic and more execution time



General Register Architecture

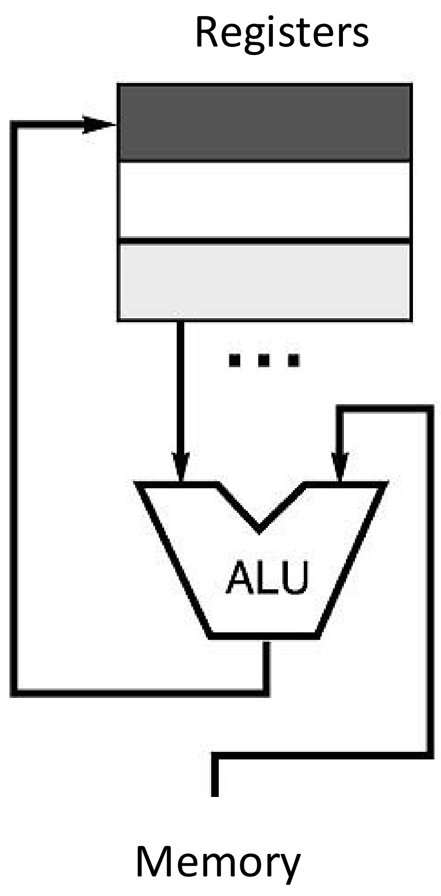
- Multiple general purpose registers (GPRs)
- Two or Three address fields in the Instruction Format
- Each address field may specify a general register or a memory word
- One operand Register and other operand Memory → Register-Memory architecture
- All operands memory → Memory-Memory architecture
- Example (3-address)
  - SUB R1, A, B which means (R1) ← mem[A] – mem[B]
  - MULT R1, R2, R3 which means (R1) ← (R2) * (R3)
- Example (2-address)
  - MULT R1, R2 which means (R1) ← (R1) * (R2)
  - ADD R1, A which means (R1) ← (R1) + mem[A]

General Register Architecture

- Example: A*B - (X+Y*Z)

3 operands	2 operands
• mul D, A, B	mov D, A
• mul E, Y, Z	mul D, B
• add E, X, E	mov E, Y
• sub E, D, E	mul E, Z
•	add E, X
•	sub E, D

- Advantages
  - Many registers are used, so program size is less
  - Requires fewer instructions (especially if 3 operands)
  - Less memory required to store the program
  - Easy to write compilers for (especially if 3 operands)
- Disadvantages
  - Very high memory traffic (especially if 3 operands)
  - Variable number of clocks per instruction
  - With two operands, more data movements are required

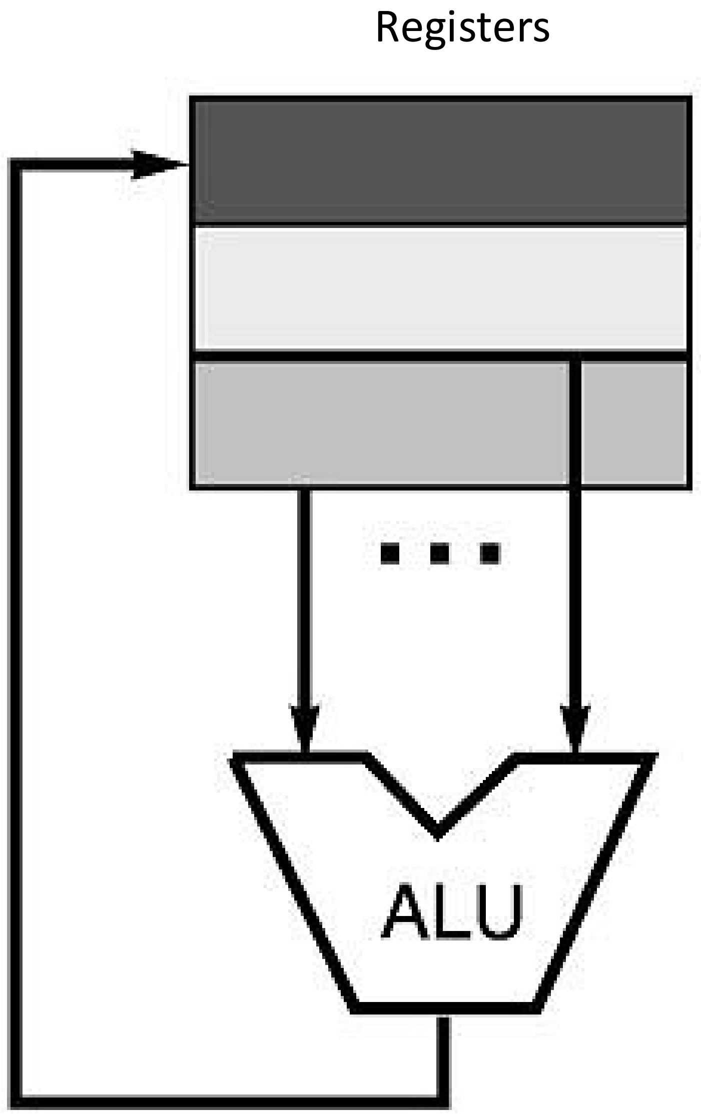


Register (Load/ Store) Architecture

- Divides instructions into two categories:
  - Memory access (Load and Store between memory and registers)
  - Arithmetic / Logic operations (which only occur between registers)
- For example, both operands and destination for an ADD operation must be in registers
- Only load and store instructions access the memory (memory indirect addressing mode)
- All other instructions use registers as operands.
- Primary motivation is speedup –registers are faster
- RISC instruction set architectures such as PowerPC, SPARC, RISC-V, ARM and MIPS are load–store architectures

Load/ Store Architecture

- Example: C = A*B - (X+Y*Z)
  - load R1, &A
  - load R2, &B
  - load R3, &X
  - load R4, &Y
  - load R5, &Z
  - mul R7, R4, R5 /* Y*Z */
  - add R8, R7, R3 /* X + Y*Z */
  - mul R9, R1, R2 /* A*B */
  - sub R10, R9, R8 /* A*B – (X+Y*Z) */
  - store R10, &C
- Advantages
  - Simple, fixed length instruction encodings
  - Instructions take similar number of cycles
  - Relatively easy to pipeline and make superscalar
- Disadvantages
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler



Stack architecture

- What is stack?
  - A portion of memory, used to store operands in successive locations
  - A data structure in which a list of data is accessed with LIFO access method
  - Only two operations: PUSH and POP
  - PUSH inserts one operand at the top of the stack
  - POP takes out one operand from the top of the stack
  - Operands are pushed or popped from one end only
  - Stack Pointer (SP) holds the address of the top of the stack

Example of PUSH and POP

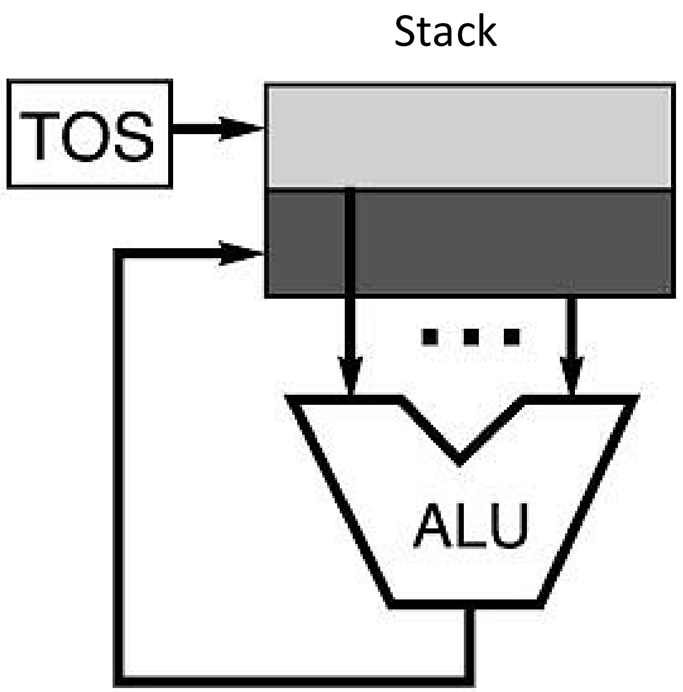
- PUSH <memory address>
  - SP ← SP – 1
  - Top of stack ← < memory address>

- POP <memory address>
  - < memory address> ← Top of stack
  - SP ← SP + 1

Stack architecture

- No operand
- This type of CPU is known as zero-address machine
- The two operands are on the top of the stack
- Result will be on the top of stack
- Example: A*B - (X+Y*Z)

```
push A
push B
mul
push X
push Y
push Z
mul
add
sub
```



Stack architecture

- Advantages
  - No address field -> length of the instruction is short
  - Low hardware requirements
  - Efficient computation of complex arithmetic expression
  - Execution of instruction is fast, because operands are stored in consecutive memory locations
  - Easy to write a simpler compiler for stack architectures

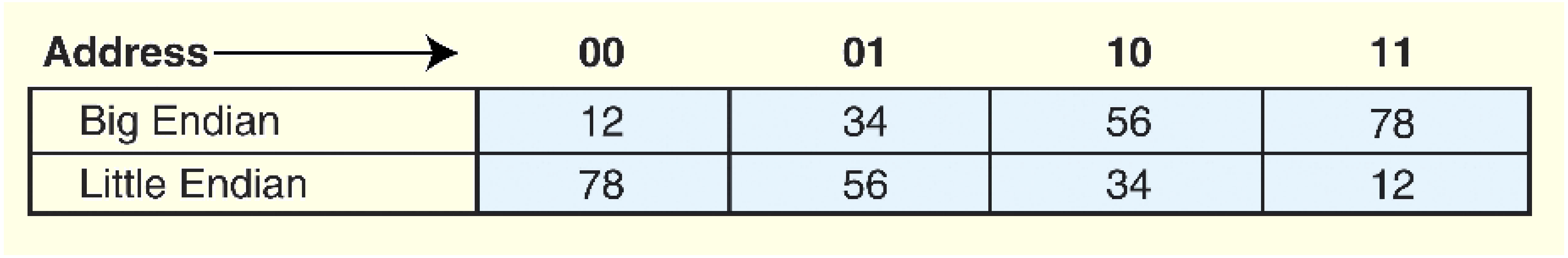
- Disadvantages
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Difficult to write an optimizing compiler for stack architectures

Ordering of bytes within a multi-byte word

- Big Endian
  - Least significant byte has highest address
  - Store the most significant byte first (at the lower address)
  - More natural
  - The sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order.
  - Example: Sun, Mac
- Little Endian
  - Least significant byte has lowest address
  - Store the most significant byte last (at the highest address)
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.
  - Example: Alphas, PCs

Example of Byte Ordering

- As an example, suppose we have the hexadecimal number 12345678.
- The big endian and small endian arrangements of the bytes are shown below.



Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

Arithmetic Expression Evaluation

- Infix notation
  - Example: (A + B) * (C + D)
- Polish Notation (or Prefix notation)
  - Example: +AB (in Prefix) means A + B (in Infix)
  - No parenthesis required
- Reverse Polish Notation (or Postfix notation)
  - Example: AB+ (in Postfix) means A + B (in Infix)
  - No parenthesis required
- Stack oriented computers are better suited to postfix notation than Infix notation
- Example: (A +B) * [C/(D-E) + F] is equivalent to AB+CDE-/F+*
- Explain with a Numerical example

Thank you

Computer Architecture
CSEN 3104
Lecture 4

Dr. Debranjan Sarkar

CISC vs. RISC Architecture

Complex Instruction Set Computer (CISC)

- More instructions for complex tasks
- Complex instructions require relatively complex processing circuits, and too much expensive
- More complex addressing modes (auto-increment, auto-decrement etc.)
- Small number of General Purpose Registers (GPR)
- Variable-length Instruction formats (spanning more than one word)
- Less suitable for pipelining

Reduced Instruction Set Computer (RISC)

- Reduced number of instructions
- This lowers processor cost, without much impact on performance.
- Simple addressing modes (Register, Register indirect etc.)
- Large register set
  - uniform (no distinction between e.g. address and data registers)
- All instructions have same length (one word)
- More suitable for pipelining

CISC vs. RISC Architecture

Complex Instruction Set Computer (CISC)

- Operands for the arithmetic / logic operations may be in the register or in memory
- Memory-to-memory data transfer possible
- Fewer instructions executed per program
- Complex instructions reduce program size but does not necessarily translate into faster execution
- Not constrained to load/store architecture. Typically use two-operand instruction format, with at least one operand in a register

Reduced Instruction Set Computer (RISC)

- Operands for the arithmetic / logic operations are always in the registers
- Memory-to-memory data transfer not possible
- More instructions executed per program
- Though the program size is more, overall execution is faster
- Instruction Set Architecture: Load/ store

CISC vs. RISC Architecture

Complex Instruction Set Computer (CISC)

- As the number of memory access is more, the impact of von Neumann bottleneck is more
- Mostly micro-programmed control units
- Example of CISC (IBM 370/168, VAX 11/780, Intel x86, PDP-11, Motorola 68000 etc.)

Reduced Instruction Set Computer (RISC)

- Reduces the impact of von Neumann bottleneck by reducing the total number of the memory access made by the CPU
- Mostly hardwired control unit
- Example of RISC (MIPS, SUN Sparc, Intel i860, Motorola 88000, IBM RS6000, PowerPC, ARM etc.)

MIPS ISA

Case Study

MIPS (Microprocessor without Interlocked Pipelined Stages)

- RISC Instruction Set Architecture
- Developed by MIPS Computer Systems -> MIPS Technologies -> Wave Computing (since December 2018)
- Multiple versions of MIPS
  - MIPS I, II, III, IV and V (32 bit)
  - MIPS32/64 (for 32- and 64-bit implementations, respectively) – six releases
- MIPS is a modular architecture supporting up to four coprocessors (CP0/1/2/3)

MIPS (Microprocessor without Interlocked Pipelined Stages)

- CP0 is the System Control Coprocessor (an essential part of the processor)
- CP1 is an optional floating point unit (FPU)
- CP2/3 are optional implementation-defined coprocessors
- For example, in the “PlayStation” video game console, CP2 is the Geometry Transformation Engine (GTE), which accelerates the processing of geometry in 3D computer graphics.
- Originally, MIPS was designed for general-purpose computing.
- During the 1980s and 1990s, MIPS processors were used for personal, workstation and server computers
- MIPS processors are used in embedded systems such as residential gateways and routers

Thank you

Computer Architecture
CSEN 3104
Lecture 5

Dr. Debranjan Sarkar

MIPS Design Principles

- Keep all instructions a single size
- Always require three register operands in arithmetic instructions
- Has only 32 registers
- PC-relative addressing for conditional branches
- Immediate addressing for constant operands

MIPS: Registers and Memory

- 32 numbers of General Purpose Registers (32-bit each) (R0 to R31)
- One 32-bit Program Counter (PC)
- 32 bit addressing capability for memory (capacity 4GB max)
- Two views of memory:
  - 2^{32} bytes with addresses 0, 1, 2, …, 2^{32}-1
  - 2^{30} 4-byte words with addresses 0, 4, 8, …, 2^{32}-4
- Both views use byte addresses
- Word address must be multiple of 4

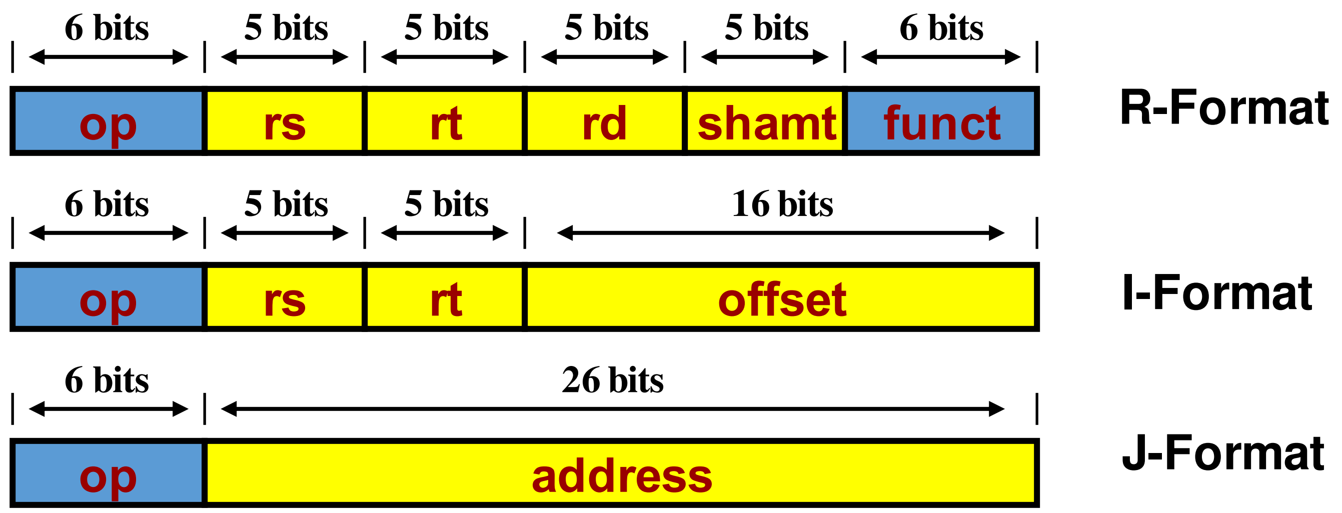
MIPS registers and usage

Each register may be referred to by number or name

Name	Register Number	Usage
$zero	0	The constant value 0
$at	1	Reserved for assembler
$v0 - $v1	2 – 3	Values for results and expression evaluation
$a0 - $a3	4 – 7	Arguments
$t0 - $t7	8 – 15	Temporary registers
$s0 - $s7	16 – 23	Saved registers
$t8 - $t9	24 – 25	More temporary registers
$k0 - $k1	26 – 27	Reserved for Operating System kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

MIPS Instructions

- All instructions 1 word = 32 bits
- 3 different formats
- Different formats for different purposes



MIPS Arithmetic & Logical Instructions

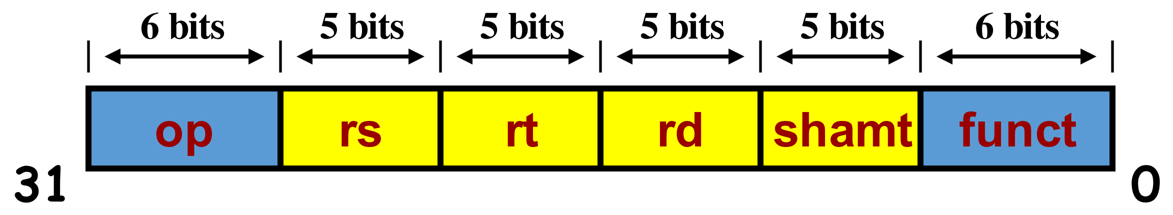
- Manipulate data in registers
  - Always 3 operands: destination + 2 sources
  - Operand order is fixed
  - Operands are always general purpose registers
  - Instruction usage (assembly)

```
add dest, src1, src2          dest=src1 + src2
```

  - Example

```
add $s1, $s2, $s3             $s1 = $s2 + $s3
or $s3, $s4, $s5              $s3 = $s4 OR $s5
sub $s1, $s2, $s3             $s1 = $s2 - $s3
and $s3, $s4, $s5             $s3 = $s4 AND $s5
```

Arithmetic & Logical Instructions (R-Format)



- Used for arithmetic, logical, shift instructions
  - op: Basic operation of the instruction (*opcode*) (Always 0 for R-Format)
  - rs: first register source operand
  - rt: second register source operand
  - rd: register destination operand
  - shamt: shift amount (0 when Not Applicable)
  - funct: function code (identifies the specific R-format instruction)

MIPS Data Transfer Instructions

- Transfer data between registers and memory
- Instruction format (assembly)

lw $dest, offset($addr)	load word
sw $src, offset($addr)	store word

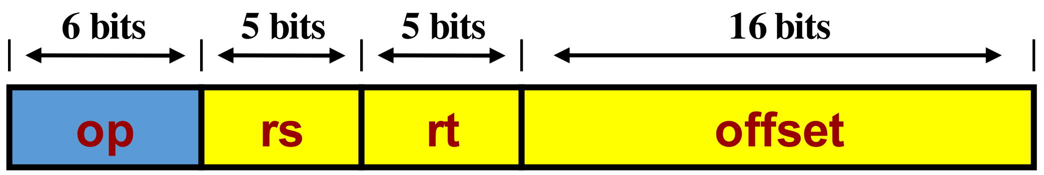
- Example

lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]
sw $s1, 100($s2)	Memory[$s2 + 100] = $s1

- Uses:
  - Accessing a variable in main memory
  - Accessing an array element

MIPS Data Transfer Instructions (I-Format)

- Transfer data between registers and memory
- Have a constant value immediately present in the instruction
- Used for load, store instructions
  - op: Basic operation of the instruction (*opcode*)
  - rs: register containing base address
  - rt: register destination/source
  - offset: 16-bit signed address offset (-32,768 to +32,767)



MIPS Branch Instructions

- Alter program flow

```
beq $s1, $s2, 25     if ($s1==$s2) PC = PC + 4 + 4*25
```

- Unconditional jump

```
j LABEL              # goto Label
```

- Conditional branches allow decision making

```
beq R1, R2, LABEL    if R1==R2 goto LABEL
bne R3, R4, LABEL    if R3!=R4 goto LABEL
```

- Example

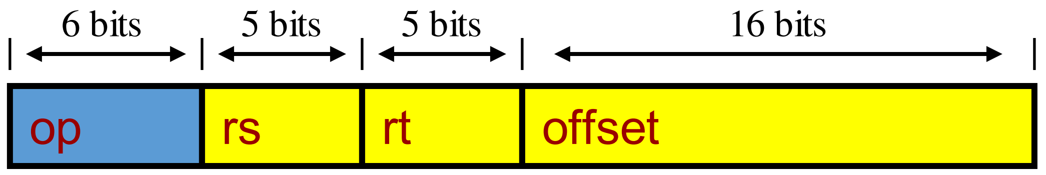
C Code

```
        if (i==j) goto L1;
        f = g + h;
L1:     f = f - i;
```

Assembly

```
        beq $s3, $s4, L1
        add $s0, $s1, $s2
L1:     sub $s0, $s0, $s3
```

MIPS Branch Instructions (I-Format)



- Branch instructions use I-Format
- Offset is added to PC when branch is effected

beq r0, r1, offset

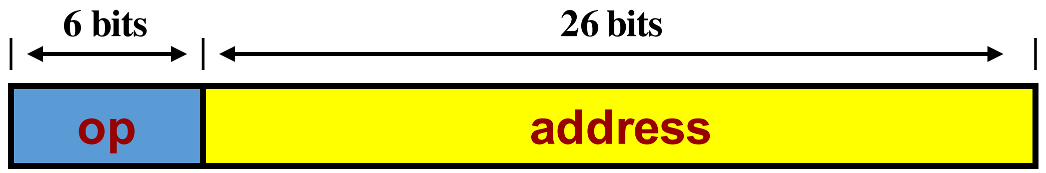
has the effect:

$$\text{if (r0==r1) pc = pc + 4 + (}\overbrace{\text{offset << 2}}^{\text{Conversion to word offset}}\text{)}$$
$$\text{else pc = pc + 4;}$$

Comparisons - What about <, <=, >, >=?

- `bne`, `beq` provide equality comparison
- `Slt dest, src1, src2` instruction sets `dest` if `src1 < src2`
-
```
slt $t0,$s3,$s4      # if $s3<$s4 $t0=1;        here $t0 is the condition register
                     # else $t0=0;
```
- Combine `Slt` with `bne` or `beq` to branch if less than
```
slt $t0,$s3,$s4             # if (a<b)
bne $t0,$zero, Less         #    goto Less;
```
- Why not include a `blt` instruction in hardware?
  - Supporting in hardware would lower performance
  - Assembler provides this function if desired (by generating the two instructions)

Jump Instructions (J-Format)



- Jump Instruction uses J-Format (`op=2`)
- The 26 bits are achieved by dropping the high-order 4 bits of the address and the low-order 2 bits
- The low order 2 bits are always 00, since addresses are always divisible by 4
- What happens during execution?

$$PC = \underbrace{PC[31:28]}_{\text{Concatenate upper 4 bits of PC to form complete 32-bit address}} : (IR[25:0] \underbrace{<< 2}_{\text{Conversion to word offset}})$$

- jal (Jump and Link) Instruction has `op=3`

Constants / Immediate Instructions

- Small constants are used quite frequently (50% of operands)

  e.g.,
  ```
  A = A + 5;
  B = B + 1;
  C = C - 18;
  ```

- MIPS Immediate Instructions (I-Format):

  ```
  addi $29, $29, 4
  slti $8, $18, 10
  ```
  } **Arithmetic instructions sign-extend immediate data**

  ```
  andi $29, $29, 6
  ori $29, $29, 4
  ```
  } **Logical instructions <u>don't</u> sign extend immediate data**

- Allows upto 16-bit constants, because
  - 16 bits fits neatly in a 32-bit instruction
  - most constants are small (i.e. < 16 bits)

- How do you load just a constant into a register?
  - ori $5, $zero, 666

MIPS Logical Instructions

- and, andi - bitwise AND
- or, ori - bitwise OR
- Example
- and $s2,$s0,$s1 $\quad$ $\$s2 \leftarrow \$s0$ AND $\$s1$
- Bitwise AND the content of register $s0 with that of $s1 and put the result in register $s2
- ori $s3,s2,252 $\quad$ $\$s3 \leftarrow \$s2$ OR 252_{10}
- Bitwise OR the content of register $s2 with 252_{10} and put the result in register $s3

Loading 32-Bit Immediate data in a register

- Normally, Immediate operations provide for 16-bit constants.
- 32-bit constant can be loaded in a register, using two instructions
- Suppose we want to load in register $t0 the value $0A50FB2F0_{16}$
- *load upper immediate - lui* (I-Format) instruction is used to set the upper 16 bits of a constant in a register
- After execution of the instruction
    - lui $t0, 1010010100001111
- The content of the register $t0 would be $0A50F0000_{16}$ (lower 16 bits filled with 0)
- Then *ori* instruction is used to fill in lower 16 bits
    - ori $t0, $t0, 1011001011110000
- After execution of this instruction
- The content of the register $t0 would be $0A50FB2F0_{16}$

MIPS Shift Instructions

- MIPS Logical Shift Instructions
  - Shift left: sll (shift-left logical) instruction
  - Right shift: srl (shift-right logical) instruction
- Example
- sll $s1,$s0,4 — $\$s1 \leftarrow \$s0 << 4$ — Shift left logical register $s0 by 4 bits and put the result in $s1
- srl $s2,$s1,8 — $\$s2 \leftarrow \$s1 >> 8$ — Shift right logical register $s1 by 8 bits and put the result in $s2

Thank You