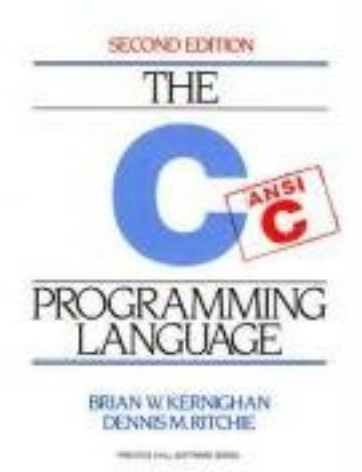
EECE 321: Computer Organization

Mohammad M. Mansour
Dept. of Electrical and Compute Engineering
American University of Beirut

Lecture 4: Machine Instructions

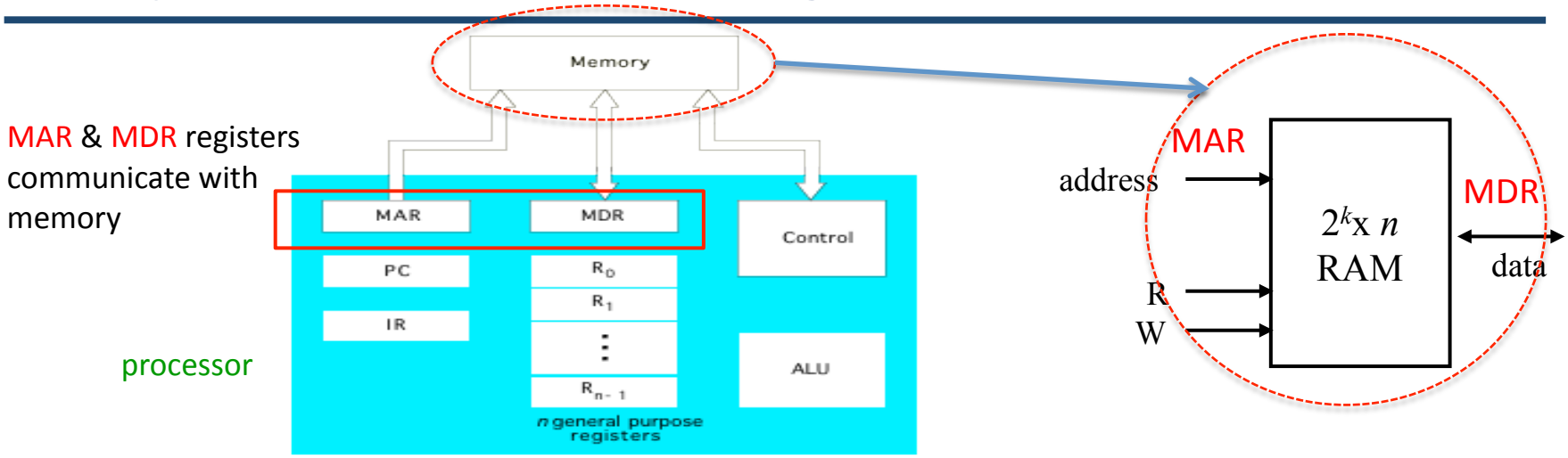
Announcements

- HW 2 to be posted
  - More questions on C/C++ programming??
- Reference: "The C Programming Language", by Kernighan and Ritchie



- Focus on bit-wise operators in C
- All HW assignments to be done individually
- Use the FOURTH edition of the textbook by Patterson and Hennessy
- Reading assignment:
  - Sections 1.1, 1.2, 1.3, 1.5, 1.6
  - Sections 2.1, 2.2

Example of Processor Internal Registers



- A processor uses registers internally to store instructions and data operands:
  - General purpose registers: R0, R1, … can be accessed by user.
    - These registers typically reside in a *Register File*
  - Special purpose register used exclusively by processor in controlling/executing instructions.
    - MDR: Memory Data Register
    - MAR: Memory Address Register
    - IR: Instruction Register
    - PC: Program Counter
- Data and instructions of a program are stored in memory externally.
  - Processor has to fetch them internally into its registers in order to operate on them.
- Remark: Don't mix up registers with external memory.

Basic Operational Concepts

- A typical instruction looks like:
  - [instruction name] [one or more operands]
  - Each instruction has a name.
  - Operands can be register addresses and/or memory addresses.
- **`Ex: Add R0,LOCA`**
  - It adds the operand at memory location LOCA to the operand in a register R0 in the processor. The result is placed in R0.
  - This is called an ADD instruction. It combines a memory access operation and an arithmetic operation.
  - Time to execute instruction = time to access memory + time to add.
- Modern microprocessors implement these 2 operations using two separate instructions: Load followed by an Add.
  - **`Load Rtemp, LOCA`**
  - **`Add R0,Rtemp`**
- A special register called Program Counter (PC) keeps track of the execution of a program. It contains the memory address of the next instruction to be fetched.
- The Instruction Register (IR) holds the instruction that is being executed.
- Communication between memory and processor is done using 2 registers:
  - Memory Address Register (MAR) holds the address of location to be accessed.
  - Memory Data Register (MDR) contains data to be written into or read out of the addressed location.

Typical Operating Steps

- Assume the instructions of a program are stored in memory (entered via Input).
  1. Execution starts when the PC is set to point to the 1st instruction of the program.
  2. The contents of PC are sent to MAR, and Read signal is sent to Memory
  3. After Memory-Access time, the addressed word (1st program inst) is loaded into MDR
  4. The contents of MDR are transferred into IR. This completes the *instruction fetch* phase; the instruction is ready to be *decoded* and *executed*.
  5. The instruction is *decoded* to determine its type and its operands.
  6. For the previous Add instruction, it is necessary to obtain the first operand from memory (*operand fetch*). Its address is sent to MAR and the operand is fetched into MDR. The second operand is supplied by the register file.
  7. Operands are forwarded to the ALU which executes the instruction.
  8. The result needs to be *stored back* either in a register or in memory. For the latter case, the address of the destination is sent to MAR and the result is sent into MDR, and a Write control signal is asserted.
  9. At some point during execution of the current instruction, the *PC is incremented* to point to the next instruction to be executed.

Summary of Steps in Executing a Typical Instruction

- Instruction fetch
- Instruction decode
- Operand fetch
- Instruction execute
- Result write-back

Instructions and Instruction Sequencing

- The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing a condition, reading a character from the keyboard, or sending a character for display on a screen.

- Instruction sequencing: Determining which instruction comes next

- A microprocessor must have instructions capable of performing 4 types of operations:
  - Data transfers between memory and the processor registers
  - ALU operations on data
  - Program sequencing and control (branches and jumps)
  - I/O transfers

- Assembly Language Notation to represent machine instructions and programs:

  INSTRUCTION_NAME OPERANDS

  - Operands are either in registers (R1, R2, …) or in memory locations MEM[Address]
  - One of the operands should be the destination where the result is stored.
  - Convention: leftmost operand is the destination.

Instructions and Instruction Sequencing (cont'd)

- Example: Assume we want to execute the following statement in C language:
  - C = A + B // A,B,C are variables assigned to distinct memory locations.
- Let the variable name (A) designate the memory address, and the content of the addressed location represent the value of the variable (MEM[A]).
- The processor can't directly operate on variables while they are in memory.
- Use a pair of loads to fetch A and B into two registers:
  - **`Load R1, A`** // fetches the contents of A into register R1; R1 ← MEM[A]
  - **`Load R2, B`** // fetches the contents of B into register R2; R2 ← MEM[B]
- Now that the operands are available in registers, the processor can add them.
  - **`Add R3,R1,R2`** **`//R3←R1+R2,assuming a 3-address add instr.`**
  - **`Add R1,R2`** **`//R1←R1+R2, assuming a 2-address add instr.`**
- After the addition operation has been carried out, the sum is available in some register R.
- This sum must be stored back in memory at location C.
- Use a store instruction:
  - **`Store R, C // stores R into memory address C; MEM[C]←R`**

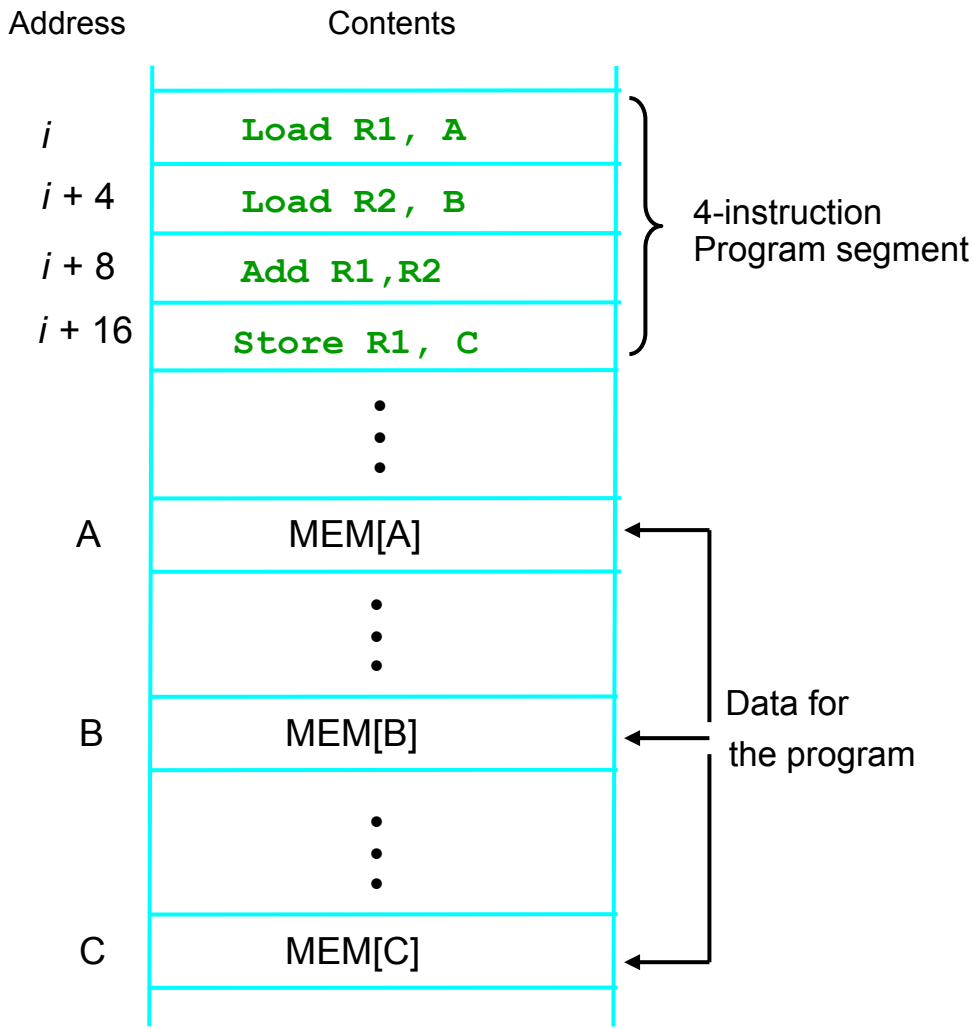
Instructions and Instruction Sequencing (cont'd)

C statement	Assembly language instructions
C = A + B	`Load R1, A`
	`Load R2, B`
	`Add R1, R2`
	`Store R1, C`

- How does the sequencing of instruction execution take place?
- Assume memory word length is 32 bits and memory is byte addressable.
- The 4 instructions are stored in successive word locations.
- The first instruction is stored in memory starting at address i.
- Since instructions are 4 bytes long, the other 3 instructions start at addresses i+4, i +8, i+16

Straight-Line Instruction Sequencing

- Register PC is initialized to address i.
- Instructions are fetched one after the other into IR for execution.
- This is referred to as straight-line sequencing.
- Note that the place where instructions are stored is separate from the place where data are stored.
- Five main phases of operations:
  - Instruction fetch
  - Instruction decode
  - Operand fetch
  - Instruction execute
  - Result write-back



Machine Instructions

- Basic job of a CPU: execute lots of instructions.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions.
- Analogy:
  - Instructions: The words of a machine's language.
  - Instruction set: The vocabulary of the language.
- The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA).
- Examples:
  - Intel 80x86 (Pentium 4)
  - IBM/Motorola PowerPC (Macintosh)
  - MIPS
  - Intel IA64 ...
- More primitive than higher level languages (HLLs)
  - e.g., no sophisticated control flow
- Very restrictive
  - e.g., MIPS Arithmetic Instructions

Instruction Set Architectures (or simply Architectures)

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX architecture had an instruction to multiply polynomials!
- Complex Instruction Set Computer (CISC)
  - Many complex instructions
  - Different instruction formats depending
    - Number of operands
    - Addressing modes
  - Corresponding micro-architecture is complex (especially pipelining)
  - EX: Intel 80x86 ISA
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – Reduced Instruction Set Computing
  - Keep the instruction set small and simple
  - This makes it easier to build fast hardware
  - Let software do complicated operations by composing simpler ones.
  - Proponents argue RISC is cheaper and faster; opponents say it puts burden on software
  - Ex: SPARC (Scalable Processor ARChitecture) by SUN Microsystems
- Modern architectures:
  - Combine attributes of both RISC and CISC flavors

MIPS Instruction Set Architecture

MIPS ISA

- MIPS
  - Microprocessor without Interlocked Pipeline Stages
  - A semiconductor company that built one of the first commercial RISC architectures
  - We will study the MIPS architecture in detail in this class
- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
  - MIPS widely used in embedded applications (e.g., NEC, Nintendo, Silicon Graphics, Sony)
  - x86 rarely used in embedded computers
  - There more embedded computers than PCs !

Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

Assembly Variables: Registers

- Unlike high-level languages like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are Registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)
- Drawback: Since registers are in hardware, they are limited
  - Solution: MIPS code must be very carefully put together to efficiently use registers
- 32 registers in MIPS
  - Why 32? Smaller is faster
- Each MIPS register is 32 bits wide
  - Groups of 32 bits form a word in MIPS
- Registers are numbered from 0 to 31
  - Each register can be referred to by number or name
  - Number references (convention):

    $0, $1, $2, … , $30, $31

Assembly Variables: Registers

- By convention, each register also has a name to make it easier to code
- For now:
  - $16 - $23 → $s0 - $s7

    (correspond to C variables)
  - $8 - $15 → $t0 - $t7

    (correspond to *temporary* variables)

- Later will explain the other 16 register names
- In general, use names to make your code more readable

Assembly Language vs. C/C++, Java

- Statements in an assembly language are instructions. They execute exactly one of a short list of simple commands.
- Unlike in C, Java (and most other High Level Languages), each line of assembly code contains at most 1 instruction.
- Comments in Assembly:
  - Hash (#) is used for MIPS comments, anything from hash mark to end of line is a comment and will be ignored
  - Not like C comments which can span multiple line /* comment */
- In C, Java (and most HLLs) variables are declared first and given a type
  - Example:

```
int fahr, celsius;
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated
  - There are no types associated with variables – the types are associated with the instructions.
- Instructions are related to operations (=, +, -, *, /) in C/C++ or Java

MIPS Addition and Subtraction

- Syntax of Instructions:

  1 2,3,4

  where:

  1) operation name

  2) operand getting result ("destination")

  3) 1st operand for operation ("source1")

  4) 2nd operand for operation ("source2")

- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity
- Addition in Assembly
  - Example: add $s0, $s1, $s2 # in MIPS
  - Equivalent to: a = b + c /* in C */

  where MIPS registers $s0, $s1, $s2 are associated with C variables a, b, c
- Subtraction in Assembly
  - Example: sub $s3, $s4, $s5 # in MIPS
  - Equivalent to: d = e - f /* in C */

  where MIPS registers $s3, $s4, $s5 are associated with C variables d, e, f

Compiling C statements into Assembly

- Compile the following C statement into MIPS Assembly
  - a = b + c + d - e;
- Break into multiple instructions:
  - add $t0, $s1, $s2 # temp = b + c
  - add $t0, $t0, $s3 # temp = temp + d
  - sub $s0, $t0, $s4 # a = temp - e
- Notice: A single line of C may break up into several lines of MIPS.
- Compile the following C statement into MIPS Assembly
  - f = -2*g;
- Use intermediate temporary registers
  - add $t0,$s1,$s1 # temp0 = 2*g
  - add $t1,$t0,$t0 # temp1 = 4*g
  - sub $s2,$t0,$t1 # f = 2*g - 4*g

What About Immediate Operands?

- One particular immediate, the number zero (0), appears very often in code.
- So we define register zero ($0 or $zero) to always have the value 0; e.g.
  - add $s0, $s1, $zero # in MIPS
  - f = g /* in C */

  where MIPS registers $s0, $s1 are associated with C variables f, g
- $zero is defined in hardware, so an instruction
  - add $zero, $zero, $s0

  will not do anything if the destination address is the register $zero!
- In general, immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:
  - addi $s0, $s1, 10 # in MIPS: add the immediate constant 10 to contents of $s1
  - f = g + 10 /* in C */

  where MIPS registers $s0, $s1 are associated with C variables f, g
- Syntax similar to add instruction, except that last argument is a number instead of a register.

Immediates

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don't include it
  - addi …, -X is equivalent to subi …, X => so no subi
- addi $s0, $s1, -10 # in MIPS
- f = g - 10 /* in C */

  where MIPS registers $s0, $s1 are associated with C variables f, g

Overflow in Arithmetic

- Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.
- Example (4-bit unsigned numbers):

+15	1111
+3	0011
------	-------
+18	10010

  - But we don't have room for 5-bit solution, so the solution would be 0010, which is +2, and wrong.
- Some languages detect overflow (Ada), some don't (like C)
- MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:
  - add (add), add immediate (addi) and subtract (sub) cause overflow to be detected
  - add unsigned (addu), add immediate unsigned (addiu) and subtract unsigned (subu) do not cause overflow detection
- Compiler selects appropriate arithmetic
- MIPS C compilers produce
  - addu, addiu, subu