CSE 31
Computer Organization

Lecture 8 – MIPS Assembly Language

Announcement

- Lab #4 this week
  - Due at 11:59pm on the same day of your next lab
  - You must demo your submission to your TA within 14 days
- HW #1 in CatCourses
  - Due Monday (2/25) at 11:59pm
- Reading assignment #1
  - Chapter 1.1 – 1.3 of zyBook
    - Do all Participation Activities in each section
    - Access through **CatCourses**
    - Due Wednesday (2/20) at 11:59pm
- Reading assignment #2
  - Chapter 2.1 – 2.9 of zyBook
    - Due Wednesday (2/27) at 11:59pm

Announcement

- Midterm exam Wednesday (3/6, not 2/27)
  - Lectures 1 – 7
  - Lab #1 - #4
  - HW #1
  - Closed book
  - 1 sheet of note (8.5” x 11”), both sides
  - Sample exam in CatCourses
  - Review session by PALS tutors next week

Assembly Language

- 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. The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA).
  - Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages), ARM, Intel IA64, RISC-V, ...

Instruction Set 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!
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – **R**educed **I**nstruction **S**et **C**omputing
  - Keep the instruction set **small** and **simple**, makes it easier to build **fast hardware**.
  - Let software do complicated operations by composing simpler ones.

MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class (also used in upper division courses)
- 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 apps, x86 little used in embedded, and more embedded computers than PCs
  - Very similar to RISC-V (open sourced), ARM

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

Assembly Variables: Registers (1/4)

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not?
    - Keep Hardware Simple
- Assembly operands are <u>registers</u>
  - Limited number of special storage 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)

Assembly Variables: Registers (2/4)

- Drawback: Since registers are in hardware, there are a predetermined number of them
  - 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 called a word in MIPS
  - Basic unit of data storage

Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:

  `$0, $1, $2, … $30, $31`

Assembly Variables: Registers (4/4)

- **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 other 16 register names
- In general, use names to make your code more readable

C, Java variables vs. registers

- In C (and most High Level Languages) variables 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; **type of instruction** determines how register contents are treated

Comments in Assembly

- Another way to make your code more readable!
- Hash (#) is used for MIPS comments
  - anything from hash mark to end of line is a comment and will be ignored
  - This is just like the C99 //
- Note: Different from C.
  - C comments have format
    `/* comment: Cannot use this with MIPS! */`
    so they can span many lines

Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands (you can search online)
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- Ok, ready for MIPS?

MIPS Addition and Subtraction (1/4)

- Syntax of Instructions:

  Format: 1,2,3,4

  where:

  1) operation by 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 and Subtraction of Integers (1/3)

- 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`

Addition and Subtraction of Integers (2/3)

- How do the following C statement work in MIPS?

```
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.
  - Notice: Everything after the hash mark on each line is ignored (comments)

Addition and Subtraction of Integers (3/3)

- How do we do this?

```
f = (g + h) - (i + j);
```

- Use intermediate temporary register

```
add $t0,$s1,$s2    # temp = g + h
add $t1,$s3,$s4    # temp = i + j
sub $s0,$t0,$t1    # f=(g+h)-(i+j)
```

Immediates

- **Immediates** are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:

  `addi $s0,$s1,10` (in MIPS)

  `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 operand is a number instead of a register.

Register Zero

- One particular immediate:
  - The number zero (0), appears very often in code (as you will see in future).
- So we define register zero (`$0` or `$zero`) to always have the value 0

  `add $s0,$s1,$zero` (in MIPS)

  `f = g` (in C)

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

  `add $zero,$zero,$s0`

  will not do anything!

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` = `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`

Quiz

1) Since there are only 8 local (`$s`) and 8 temp (`$t`) variables, we can't write MIPS for C expressions that contain > 16 vars.

2) If `p` (stored in `$s0`) is a pointer to an array of `int`s, then `p++;` would be `addi $s0 $s0 1`

	12
a)	FF
b)	FT
c)	TF
d)	TT
e)	dunno

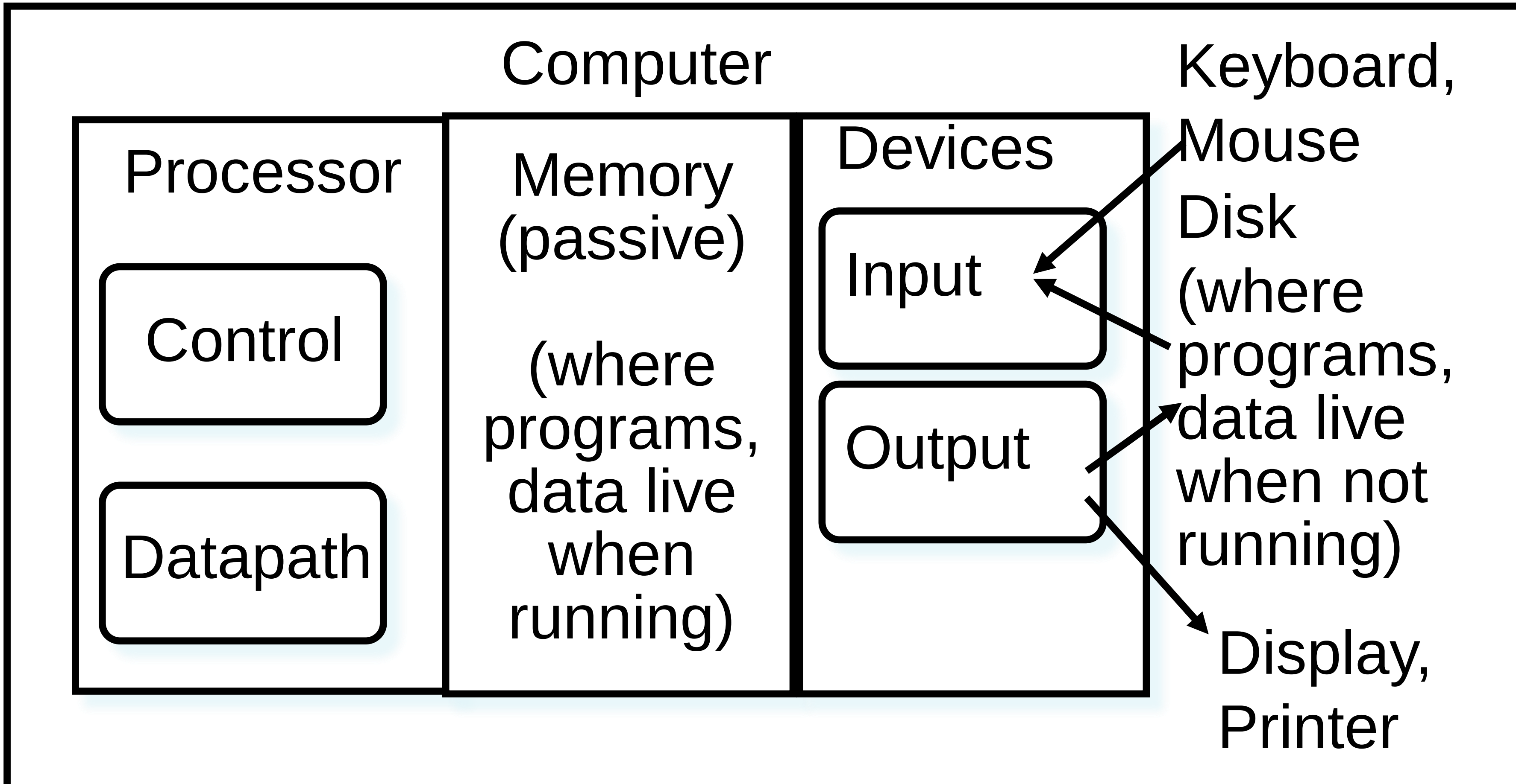
Quiz

1) Since there are only 8 local (`$s`) and 8 temp (`$t`) variables, we can't write MIPS for C exprs that contain > 16 vars.

2) If `p` (stored in `$s0`) is a pointer to an array of `int`s, then `p++;` would be `addi $s0 $s0 1`

	12
a)	**FF**
b)	FT
c)	TF
d)	TT
e)	dunno

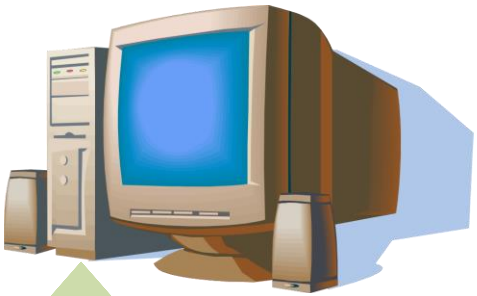
Five Components of a Computer



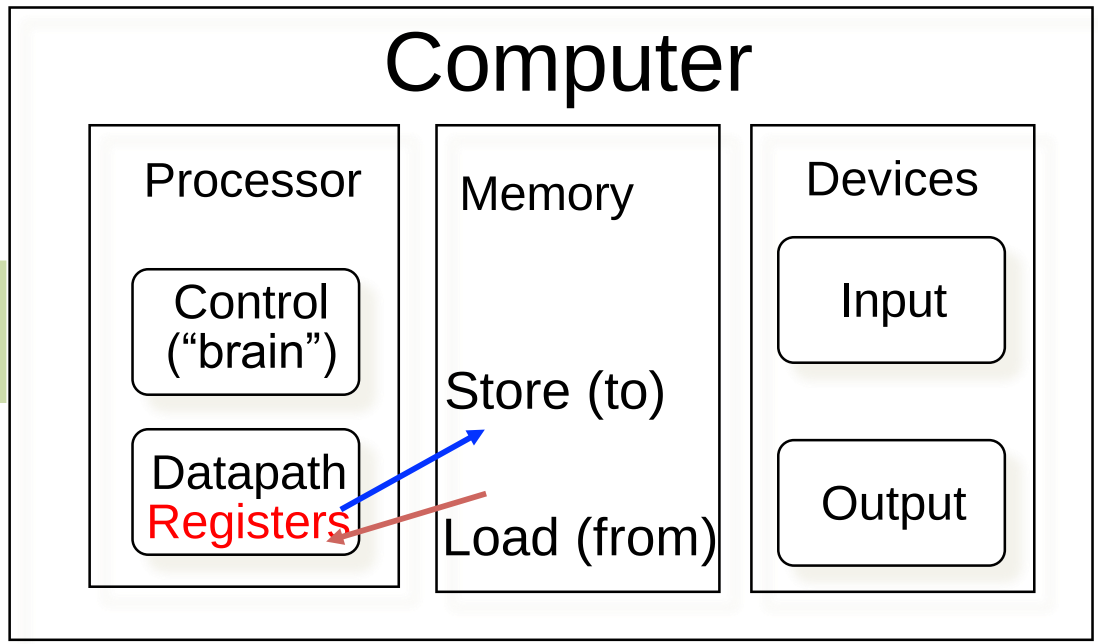
Assembly Operands: Memory

- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- Data transfer instructions transfer data between registers and memory:
  - Memory to register
  - Register to memory

Anatomy: 5 components of any Computer



Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.



These are "data transfer" instructions…

Data Transfer: Memory to Reg (1/4)

- To transfer a word of data, we need to specify two things:
  - Register: specify this by # ($0 - $31) or symbolic name ($s0,…,$t0,…)
  - Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - Other times, we want to be able to offset from this pointer.

- Remember: "Load FROM memory"

Data Transfer: Memory to Reg (2/4)

- To specify a memory address to load from, specify two things:
  - A register containing a pointer to memory
  - A numerical offset (in bytes), how far away from the address
- The desired memory address is the sum of these two values.
- Example: `8($t0)`
  - specifies the memory address pointed to by the value in `$t0`, plus 8 bytes

Data Transfer: Memory to Reg (3/4)

- Load Instruction Syntax:

  Format: 1,2,3(4)

  - where

    1) operation name

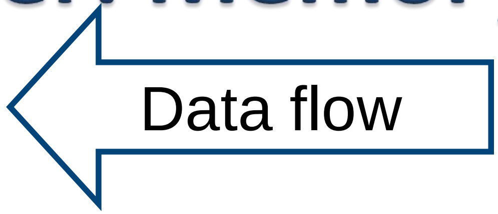
    2) register that will receive value

    3) numerical offset in bytes

    4) register containing pointer to memory

- MIPS Instruction Name:
  - `lw` (meaning **Load Word**, so 32 bits (one word) are loaded at a time)

Data Transfer: Memory to Reg (4/4)



Example: `lw $t0,12($s0)`

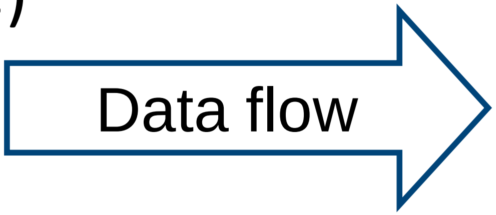
This instruction will take the pointer stored in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0

- Notes:
  - `$s0` is called the base register
  - 12 is called the offset
  - offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure (note offset must be a constant known at assembly time)

Data Transfer: Reg to Memory

- Also want to store from register into memory
  - Store instruction syntax is identical to Load's
- MIPS Instruction Name:

  `sw` (meaning Store Word, so 32 bits or one word is stored at a time)



- Example: `sw $t0,12($s0)`

  This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into that memory address
- Remember: "Store INTO memory"