CSE 31
Computer Organization

Lecture 14 – MIPS Assembly Language (contd)

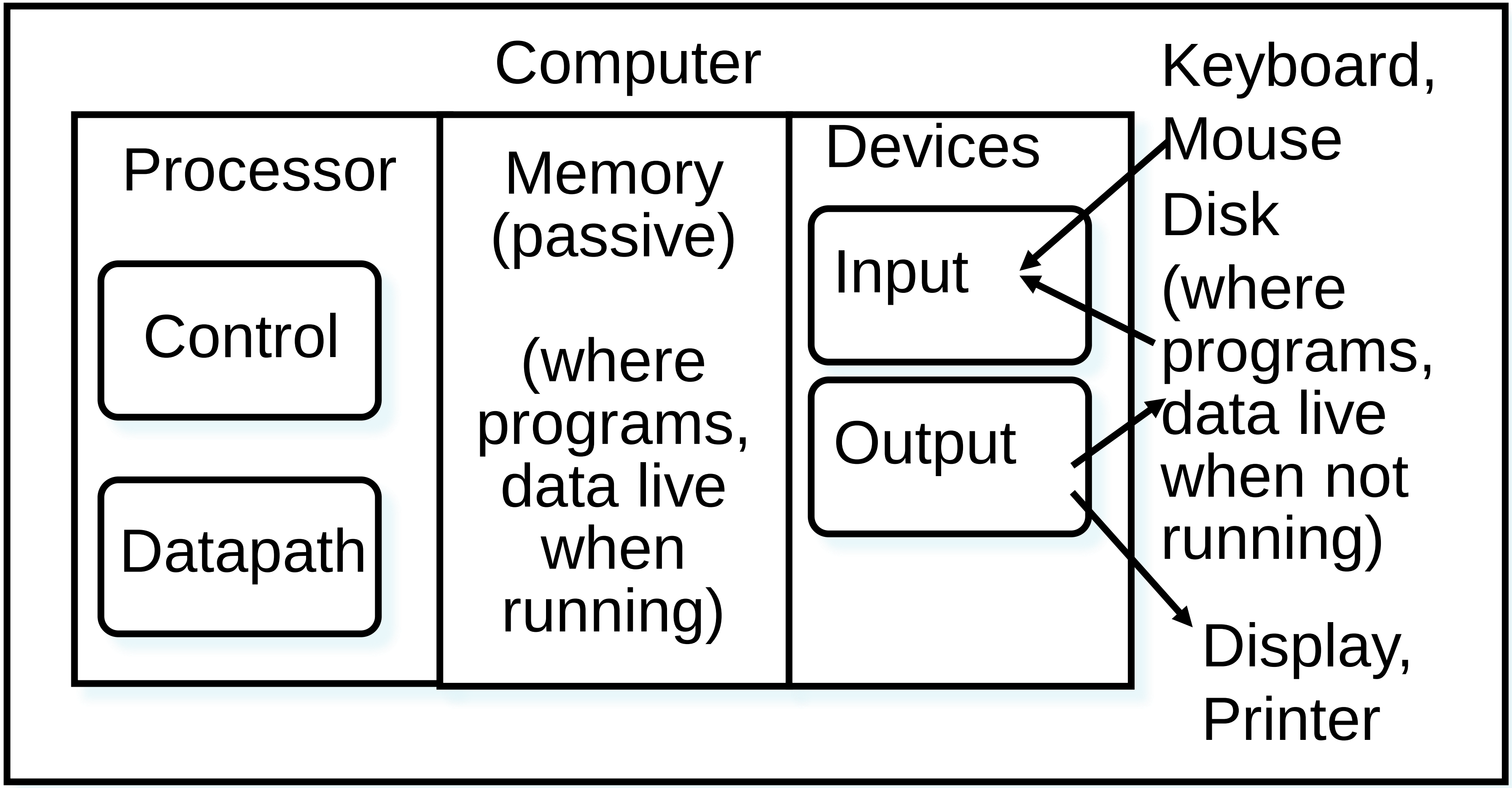
Announcements

- Labs
  - Lab 5 grace period ends ***next* week**
    - » No penalty for submission during grace period
    - » Demo is REQUIRED to receive full credit
  - Lab 6 out this week
    - » Due at 11:59pm on the same day of your lab after next (with **14 days grace period** after due date)
    - » You must demo your submission to your TA within 21 days from posting of lab
    - » Demo is REQUIRED to receive full credit
  - Lab 7 and Project 02 out next week
- Reading assignments
  - Reading 04 (zyBooks 4.1 – 4.9) due 20-MAR and Reading 05 (zyBooks 1.6 - 1.7, 6.1 - 6.3) due 03-APR
    - » Complete **Participation Activities** in each section to receive grade
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses
- Homework assignment
  - Homework 03 (zyBooks 3.1 – 3.7, 3.9) due **tonight**, 13-MAR
    - » Complete **Challenge Activities** in each section to receive grade
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

Announcements

- Project 01
  - **Due 17-MAR**
  - Can work in teams of 2 students
    - » Each team member must identify teammate in "Comments…" text-box at the submission page
    - » If working in teams, each student must submit code (can be the same as teammate) and demo individually
    - » Grade can vary among teammates depending on demo
  - Demo required for project grade
    - » No partial credit for submission without demo
  - **No grace period**
    - » **Must complete submission and demo by due date.**
- Extra office hours to facilitate Project 01 demos posted on CatCourses

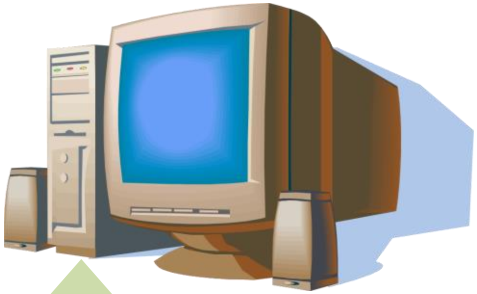
Five Components of a Computer



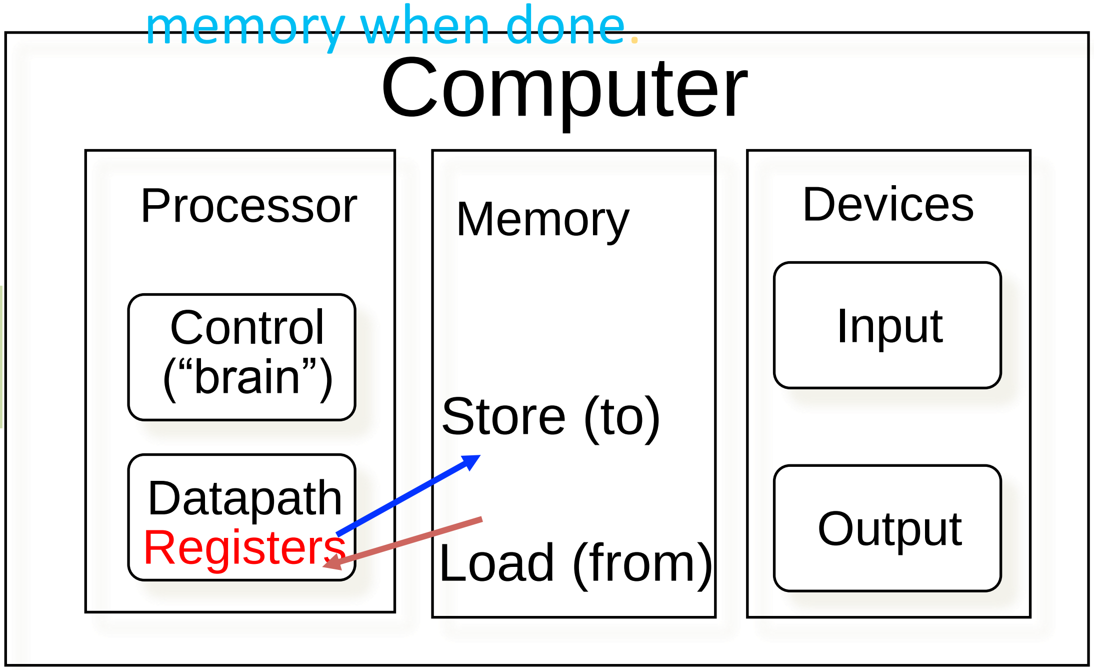
Assembly Operands: Memory

- 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.



Computer

Processor

Control ("brain")

Datapath
Registers

Memory

Store (to)

Load (from)

Devices

Input

Output

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

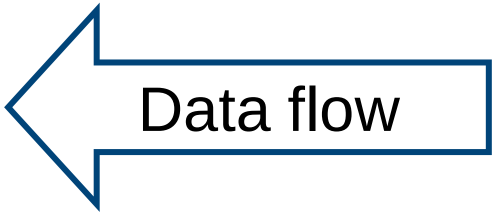
    2) register that will receive value

    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)


Data flow

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

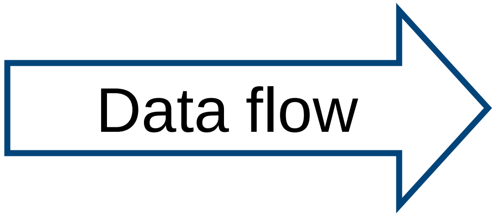
This instruction will take the pointer stored in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0

- Notes:
  - `$s0` is called the base register
  - 12 is called the offset
  - offset is generally used in accessing elements of array or structure: base register 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”

Pointers vs. Values

- Key Concept: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory addr), and so on
  - E.g., If you write: `add $t2,$t1,$t0 # c = b + A;` then `$t0` and `$t1` better contain values that can be added
  - E.g., If you write:

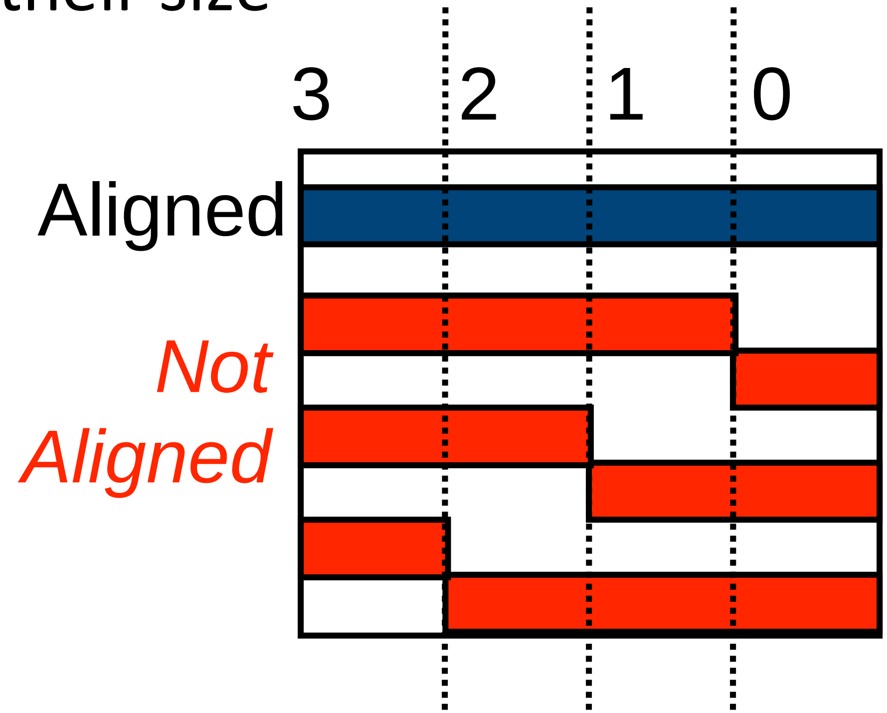
    ```
    lw $t2, 0($t0) # c = A[0];
    add $t2, $t2, $t1 # c = A[0] + b
    ```

    then `$t0` better contain a pointer

- Don’t mix these up!

More Notes about Memory: Alignment

- MIPS requires that all *words* start at byte addresses that are multiples of 4 bytes
- Called Alignment: objects fall on address that is multiple of their size



Last hex digit of address is:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

Additional Resources:
Wikipedia Article
GeeksforGeeks Article
Stack overflow Response

Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many assembly language programmers have toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  - Also, remember that for both `lw` and `sw`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common variables in memory: spilling
- Why not keep all variables in memory?
  - Smaller is faster:
    - » Registers are faster than memory
  - Registers more versatile:
    - » MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - » MIPS data transfer only read or write 1 operand per instruction, and no operation

Compilation with Memory

- What offset in `lw` to select `A[5]` in C?
  - 4 x 5 = 20 to select `A[5]`: byte vs. word
- Compile by hand using registers:

```
g = h + A[5];
```

g: `$s1`, h: `$s2`, `$s3`: base address of `A`

- 1st transfer from memory to register:

```
lw   $t0, 20($s3) # $t0 gets A[5]
```

  - Add 20 to `$s3` to select `A[5]`, put into `$t0`
- Next add it to h and place in `g`

```
add $s1,$s2,$t0  # $s1 = h + A[5]
```

Quiz

We want to translate `*x = *y` into MIPS

(`x`, `y` ptrs stored in: `$s0` `$s1` Which of the following statements (or sequence of statements) will achieve this?

```
1: add $s0,   $s1, zero
2: add $s1,   $s0, zero
3: lw  $s0, 0($s1)
4: lw  $s1, 0($s0)
5: lw  $t0, 0($s1)
6: sw  $t0, 0($s0)
7: lw  $s0, 0($t0)
8: sw  $s1, 0($t0)
```

```
a)  1 or 2
b)  3 or 4
c)  5 → 6
d)  6 → 5
e)  7 → 8
```

Quiz

We want to translate `*x = *y` into MIPS

(`x`, `y` ptrs stored in: `$s0` `$s1` Which of the following statements (or sequence of statements) will achieve this?

```
1:  add $s0,    $s1, zero
2:  add $s1,    $s0, zero
3:  lw  $s0, 0($s1)
4:  lw  $s1, 0($s0)
5:  lw  $t0, 0($s1)
6:  sw  $t0, 0($s0)
7:  lw  $s0, 0($t0)
8:  sw  $s1, 0($t0)
```

```
a)  1 or 2
b)  3 or 4
c)  5 → 6
d)  6 → 5
e)  7 → 8
```

Summary

- In MIPS Assembly Language:
  - Registers replace variables
  - One Instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster

- New Instructions:

  `add, addi, sub`

- New Registers:

  C Variables: `$s0` - `$s7`

  Temporary Variables: `$t0` - `$t7`

  Zero: `$zero`

So Far...

- All instructions so far only manipulate data…we've built a calculator of sorts.

- In order to build a computer, we need ability to make decisions…

- C (and MIPS) provide labels to support "goto" jumps to places in code.
  - C: Horrible style;
  - MIPS: Necessary!

C Decisions: `if` Statements

- 2 kinds of if-statements in C

  `if` *(condition) clause*

  `if` *(condition) clause1* `else` *clause2*

- Rearrange 2nd if-statement as shown below:

```
if    (condition) goto L1;
      clause2;
      goto L2;
L1:   clause1;
L2:
```

- Not as elegant as if-else, but same meaning