inst.eecs.berkeley.edu/~cs61c

UCB CS61C : Machine Structures

Lecture 9 – Introduction to MIPS Data Transfer & Decisions I

Lecturer SOE Dan Garcia

Hi to **Nick Carlson** from **U Northern Colorado**

2008-02-11

VIDEOPHILIA?

Study finds more and more people prefer to stay indoors with computers & TVs. This has been termed "*videophilia*", and has been shown to be a cause for "obesity, lack of socialization, attention disorders and poor academic performance". Take a walk, folks!

www.sfgate.com/cgi-bin/article.cgi?f=/c/a/2008/02/10/MNHUURU6R.DTL

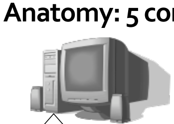
Review

- 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`

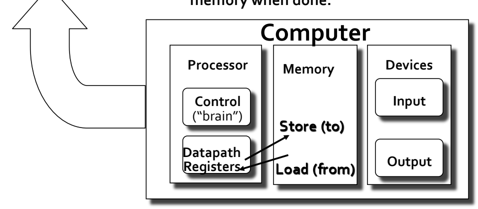
Assembly Operands: Memory

- 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 copy from, specify two things:
  - A register containing a pointer to memory
  - A numerical offset (in bytes)
- 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:

  `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 or 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 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 <u>base register</u>
  - `12` is called the <u>offset</u>
  - 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)

  Data flow
- 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 v. 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,$t0` then `$t0` and `$t1` better contain values that can be added
  - E.g., If you write: `lw $t2,0($t0)` then `$t0` better contain a pointer
- Don't mix these up!

Addressing: Byte vs. Word

- Every word in memory has an <u>address</u>, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
  - `Memory[0], Memory[1], Memory[2], ...`

    Called the "<u>address</u>" of a word
- Computers needed to access 8-bit <u>bytes</u> as well as words (4 bytes/word)
- Today machines address memory as bytes, (i.e., "Byte Addressed") hence 32-bit (4 byte) word addresses differ by 4
  - `Memory[0],Memory[4],Memory[8]`

Compilation with Memory

- What offset in `lw` to select `A[5]` in C?
- 4x5=20 to select `A[5]`: byte v. 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 <u>20</u> to `$s3` to select `A[5]`, put into `$t0`
- Next add it to `h` and place in `g`

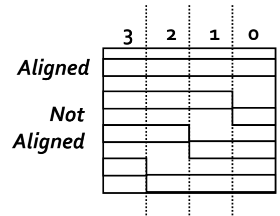
  `add $s1,$s2,$t0 # $s1 = h+A[5]`

Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many an assembly language programmer has 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)

More Notes about Memory: Alignment

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes



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}

- Called <u>Alignment</u>: objects fall on address that is multiple of their size

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: <u>spilling</u>
- 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

Administrivia

- Project 1 due this Sat (not Sun) @ 11:59pm
- Great talk: Vincent Cerf, 5-6:30pm,
- Faux exam tonight @ 7pm-9pm in 306 Soda
  - We give you an actual exam from old Qs
  - You take it for an hour
  - You hand it your neighbor
  - A superstar TA will walk in and take it
- Other administrivia?

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 <u>labels</u> to support "`goto`" jumps to places in code.
  - C: Horrible style; MIPS: Necessary!
- Heads up: pull out some papers and pens, you'll do an in-class exercise!

C Decisions: `if` Statements

- 2 kinds of `if` statements in C
  - `if (`*condition*`)` *clause*
  - `if (`*condition*`)` *clause1* `else` *clause2*
- Rearrange 2nd `if` into following:

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

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

MIPS Decision Instructions

- Decision instruction in MIPS:

```
beq   register1, register2, L1
```

`beq` is "Branch if (registers are) equal"
Same meaning as (using C):

```
if  (register1==register2) goto L1
```

- Complementary MIPS decision instruction

```
bne   register1, register2, L1
```

`bne` is "Branch if (registers are) not equal"
Same meaning as (using C):

```
if  (register1!=register2) goto L1
```

- Called <u>conditional branches</u>

MIPS Goto Instruction

- In addition to conditional branches, MIPS has an <u>unconditional branch</u>:

```
j label
```

- Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition
- Same meaning as (using C): `goto label`
- Technically, it's the same effect as:

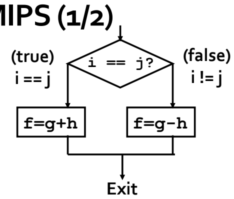
```
beq $0,$0,label
```

since it always satisfies the condition.

Compiling C `if` into MIPS (1/2)

- Compile by hand

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



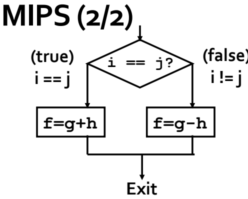
- Use this mapping:

f: `$s0`
g: `$s1`
h: `$s2`
i: `$s3`
j: `$s4`

Compiling C `if` into MIPS (2/2)

- Compile by hand

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



- Final compiled MIPS code:

```
      beq $s3,$s4,True   # branch i==j
      sub $s0,$s1,$s2    # f=g-h(false)
      j   Fin            # goto Fin
True: add $s0,$s1,$s2    # f=g+h (true)
Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches). Generally not found in HLL code.

Peer Instruction

We want to translate `*x = *y` into MIPS
(`x`, `y` ptrs stored in: `$s0 $s1`)

```
A: add $s0,   $s1, zero
B: add $s1,   $s0, zero
C: lw  $s0, 0($s1)
D: lw  $s1, 0($s0)
E: lw  $t0, 0($s1)
F: sw  $t0, 0($s0)
G: lw  $s0, 0($t0)
H: sw  $s1, 0($t0)
```

```
0: A
1: B
2: C
3: D
4: E→F
5: E→G
6: F→E
7: F→H
8: H→G
9: G→H
```


"And in Conclusion…"

- Memory is byte-addressable, but `lw` and `sw` access one word at a time.
- A pointer (used by `lw` and `sw`) is just a memory address, we can add to it or subtract from it (using offset).
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using conditional statements within `if`, `while`, `do while`, `for`.
- MIPS Decision making instructions are the conditional branches: `beq` and `bne`.
- New Instructions:

```
lw, sw, beq, bne, j
```