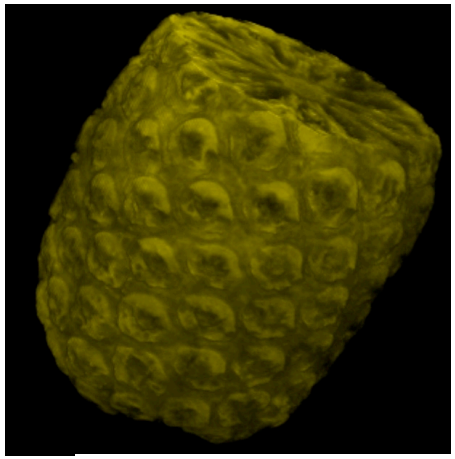
inst.eecs.berkeley.edu/~cs61c

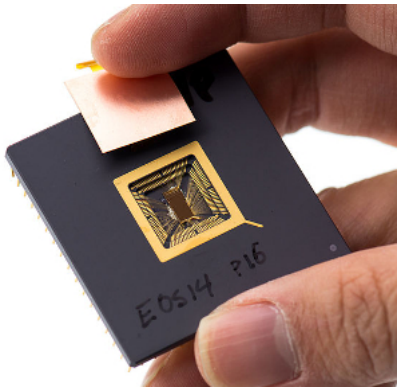
CS61C : Machine Structures

Lecture 7 – Introduction to MIPS Decisions II

2014-09-15



Instructor:
Miki Lustig



EETimes article 08/07/2014
RISC-V: An open standard for SoCs. The case for an open ISA (Krste,Patterson, UC Berkeley).
While the likely first beachhead for RISC-V is the IoT, our ambitious goal is grander: Just as Linux has become the standard OS for most computing devices, we envision RISC-V becoming the standard ISA for all computing devices.

http://www.eetimes.com/author.asp?section_id=36&doc_id=1323406&

ANGEL

- http://riscv.org/angel/

Review

- 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, so 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
```

Loading, Storing bytes 1/2

- In addition to word data transfers (lw, sw), MIPS has byte data transfers:
  - load byte: `lb`
  - store byte: `sb`
- same format as `lw`, `sw`
- E.g., `lb $s0, 3($s1)`
  - contents of memory location with address = sum of "3" + contents of register *s1* is copied to the low byte position of register *s0*.

Loading, Storing bytes 2/2

- What do with other 24 bits in the 32 bit register?
  - lb: sign extends to fill upper 24 bits

- Normally don't want to sign extend chars
- MIPS instruction that doesn't sign extend when loading bytes:
  - load byte unsigned: `lbu`

Overflow in Arithmetic (1/2)

- 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".

Overflow in Arithmetic (2/2)

- Some languages detect overflow (Ada), some don't (most C implementations)
- MIPS solution is 2 kinds of arithmetic instructs:
  - These cause overflow to be detected
    - add (`add`)
    - add immediate (`addi`)
    - subtract (`sub`)
  - These do not cause overflow detection
    - add unsigned (`addu`)
    - add immediate unsigned (`addiu`)
    - subtract unsigned (`subu`)
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce `addu, addiu, subu`

Two "Logic" Instructions

- Here are 2 more new instructions
- Shift Left: `sll $s1,$s2,2 #s1=s2<<2`
  - Store in `$s1` the value from `$s2` shifted 2 bits to the left (they fall off end), inserting 0's on right; << in C.
  - Before: $0000\ 0002_{hex}$
    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$
  - After: $0000\ 0008_{hex}$
    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_{two}$
  - What arithmetic effect does shift left have?
- Shift Right: `srl` is opposite shift; >>

Loops in C/Assembly (1/3)

- Simple loop in C; `A[]` is an array of ints

```
do { g = g + A[i];
     i = i + j;
} while (i != h);
```

- Rewrite this as:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

- Use this mapping:

```
  g,   h,   i,   j, &A[0]
$s1, $s2, $s3, $s4, $s5
```

Loops in C/Assembly (2/3)

- Final compiled MIPS code:

```
Loop: sll  $t1,$s3,2     # $t1= 4*i
      addu $t1,$t1,$s5   # $t1=addr A+4i
      lw   $t1,0($t1)    # $t1=A[i]
      addu $s1,$s1,$t1   # g=g+A[i]
      addu $s3,$s3,$s4   # i=i+j
      bne  $s3,$s2,Loop  # goto Loop
                         # if i!=h
```

- Original code:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

Loops in C/Assembly (3/3)

- There are three types of loops in C:
  - `while`
  - `do … while`
  - `for`
- Each can be rewritten as either of the other two, so the method used in the previous example can be applied to these loops as well.
- Key Concept: Though there are multiple ways of writing a loop in MIPS, the key to decision-making is conditional branch

Inequalities in MIPS (1/4)

- Until now, we've only tested equalities (== and != in C). General programs need to test < and > as well.
- Introduce MIPS Inequality Instruction:
  - "Set on Less Than"
  - Syntax: `slt reg1,reg2,reg3`
  - Meaning: `reg1 = (reg2 < reg3);`

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```

Same thing…

"set" means "change to 1",
"reset" means "change to 0".

Inequalities in MIPS (2/4)

- How do we use this? Compile by hand:

```
if (g < h) goto Less; #g:$s0, h:$s1
```

- Answer: compiled MIPS code…

```
slt $t0,$s0,$s1  # $t0 = 1 if g<h
bne $t0,$0,Less  # goto Less
                 # if $t0!=0
                 # (if (g<h)) Less:
```

- Register `$0` always contains the value 0, so `bne` and `beq` often use it for comparison after an `slt` instruction.

- A `slt` → `bne` pair means `if(… < …)goto…`

Inequalities in MIPS (3/4)

- Now we can implement <, but how do we implement >, ≤ and ≥ ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement ≤ in one or more instructions using just `slt` and branches?
  - What about >?
  - What about ≥?

Inequalities in MIPS (4/4)

```
# a:$s0,  b:$s1
 slt $t0,$s0,$s1 # $t0 = 1 if a<b
 beq $t0,$0,skip # skip if a >= b
     <stuff> # do if a<b
 skip:
```

Two independent variations possible:

- Use `slt $t0,$s1,$s0` instead of `slt $t0,$s0,$s1`
- Use `bne` instead of `beq`

Immediates in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`
  - Helpful in for loops

C

```
if (g >= 1) goto Loop
```

MIPS

```
Loop:   . . .
  slti $t0,$s0,1     # $t0 = 1 if
                     # $s0<1 (g<1)
  beq  $t0,$0,Loop   # goto Loop
                     # if $t0==0
                     # (if (g>=1))
```

An `slt` → `beq` pair means `if(… ≥ …)goto…`

What about unsigned numbers?

- Also unsigned inequality instructions:

  `sltu`, `sltiu`

  …which sets result to 1 or 0 depending on unsigned comparisons

- What is value of `$t0`, `$t1`?

  (`$s0` = `FFFF FFFA`$_{hex}$, `$s1` = `0000 FFFA`$_{hex}$)

```
slt $t0, $s0, $s1
sltu $t1, $s0, $s1
```

MIPS Signed vs. Unsigned – diff meanings!

- MIPS terms Signed/Unsigned "overloaded":
  - Do/Don't sign extend
    - `(lb, lbu)`
  - Do/Don't overflow
    - `(add, addi, sub, mult, div)`
    - `(addu, addiu, subu, multu, divu)`
  - Do signed/unsigned compare
    - `(slt, slti/sltu, sltiu)`

Peer Instruction

```
Loop:addi $s0 $s0       # i = i - 1
     slti $t0 $s1       # $t0 = (j < 2)
     beq  $t0 $0  Loop  # goto Loop if $t0 == 0
     slt  $t0 $s1 $s0   # $t0 = (j < i)
     bne  $t0 $0  Loop  # goto Loop if $t0 != 0
```

($s0=i $s1=j)

What C code properly fills in the blank in loop below?

```
do {i--;} while(__);
```

a)	j	<	2	&&	j	<	i
a)	j	≥	2	&&	j	<	i
b)	j	<	2	&&	j	≥	i
b)	j	≥	2	&&	j	≥	i
c)	j	>	2	&&	j	<	i
c)	j	<	2	||	j	<	i
d)	j	≥	2	||	j	≥	i
d)	j	<	2	||	j	≥	i
e)	j	≥	2	||	j	<	i
e)	j	>	2	||	j	<	i

"And in conclusion…"

- To help the conditional branches make decisions concerning inequalities, we introduce: "Set on Less Than" called
  `slt, slti, sltu, sltiu`
- One can store and load (signed and unsigned) bytes as well as words with `lb, lbu`
- Unsigned add/sub don't cause overflow
- New MIPS Instructions:
  `sll, srl, lb, lbu`
  `slt, slti, sltu, sltiu`
  `addu, addiu, subu`