

Lecture #9 – MIPS Procedures II, Logical & Shift Ops, and Instruction Representation I

2008-7-7



Albert Chae, Instructor

Review

- Functions called with `jal`, return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just be sure to leave it the way you found it.
- Instructions we know so far
 - Arithmetic: `add`, `addi`, `sub`, `addu`, `addiu`, `subu`
 - Memory: `lw`, `sw`, `lb`, `sb`, `lbu`
 - Decision: `beq`, `bne`, `slt`, `slti`, `sltu`, `sltiu`
 - Unconditional Branches (Jumps): `j`, `jal`, `jr`
- Registers we know so far
 - All of them!
 - There are CONVENTIONS when calling procedures!



Register Conventions (1/4)

- **Caller**: the calling function
- **Callee**: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- **Register Conventions**: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed.



Register Conventions (2/4) - saved

- `$0`: **No Change**. Always 0.
- `$s0-$s7`: **Restore if you change**. Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- `$sp`: **Restore if you change**. The stack pointer must point to the same place before and after the `jal` call, or else the caller won't be able to restore values from the stack.
- **HINT** -- All saved registers start with **S**!



Register Conventions (3/4) - volatile

- `$ra`: **Can Change**. The `jal` call itself will change this register. Caller needs to save on stack if nested call.
- `$v0-$v1`: **Can Change**. These will contain the new returned values.
- `$a0-$a3`: **Can change**. These are volatile argument registers. Caller needs to save if they'll need them after the call.
- `$t0-$t9`: **Can change**. That's why they're called temporary; any procedure may change them at any time. Caller needs to save if they'll need them afterwards.



Register Conventions (4/4)

- What do these conventions mean?
 - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a `jal` call.
 - Function E must save any S (saved) registers it intends to use before garbling up their values
- Remember: Caller/callee need to save only temporary/saved registers they are using, not all registers.



Parents leaving for weekend analogy (1/5)

- Parents (**main**) leaving for weekend
- They (**caller**) give keys to the house to kid (**callee**) with the rules (**calling conventions**):
 - You can trash the **temporary room(s)**, like the **den and basement (registers)** if you want, we don't care about it
 - **BUT** you'd better leave the rooms (**registers**) that we want to **save** for the guests untouched. **"these rooms better look the same when we return!"**



Who hasn't heard this in their life?

CS61C L9 MIPS Proc II, Logical & Shift Ops, and Instruction Representation I (7)

Chae, Summer 2008 ©UCB

Parents leaving for weekend analogy (2/5)

- Kid now "owns" rooms (**registers**)
- Kid wants to use the **saved** rooms for a wild, wild party (**computation**)
- What does kid (**callee**) do?
 - Kid takes what was in these rooms and puts them in the garage (**memory**)
 - Kid throws the party, **trashes everything** (except garage, who goes there?)
 - Kid restores the rooms the parents wanted **saved after the party** by **replacing the items from the garage (memory) back into those saved rooms**



CS61C L9 MIPS Proc II, Logical & Shift Ops, and Instruction Representation I (8)

Chae, Summer 2008 ©UCB

Parents leaving for weekend analogy (3/5)

- Same scenario, except **before** parents return and kid replaces **saved** rooms...
- Kid (**callee**) has left valuable stuff (**data**) all over.
 - Kid's friend (**another callee**) wants the house for a party when the **kid** is away
 - Kid knows that friend might **trash the place** destroying valuable stuff!
 - Kid remembers rule parents taught and now becomes the "heavy" (**caller**), instructing friend (**callee**) on good rules (**conventions**) of house.



CS61C L9 MIPS Proc II, Logical & Shift Ops, and Instruction Representation I (9)

Chae, Summer 2008 ©UCB

Parents leaving for weekend analogy (4/5)

- If kid had data in **temporary rooms** (which were going to be trashed), there are three options:
 - Move items directly to garage (**memory**)
 - Move items to **saved rooms** whose contents have already been moved to the garage (**memory**)
 - Optimize lifestyle (**code**) so that the amount you've got to shlep stuff back and forth from garage (**memory**) is minimized
 - Mantra: "Minimize register footprint"



Otherwise: "Dude, where's my data?!"

CS61C L9 MIPS Proc II, Logical & Shift Ops, and Instruction Representation I (10)

Chae, Summer 2008 ©UCB

Parents leaving for weekend analogy (5/5)

- **Friend** now "owns" rooms (**registers**)
- Friend wants to use the **saved** rooms for a wild, wild party (**computation**)
- What does friend (**callee**) do?
 - Friend takes what was in these rooms and puts them in the garage (**memory**)
 - Friend throws the party, **trashes everything** (except garage)
 - Friend restores the rooms the kid wanted **saved after the party** by **replacing the items from the garage (memory) back into those saved rooms**



CS61C L9 MIPS Proc II, Logical & Shift Ops, and Instruction Representation I (11)

Chae, Summer 2008 ©UCB

Bitwise Operations

- Up until now, we've done arithmetic (add, sub, addi), memory access (lw and sw), and branches and jumps.
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- **New Perspective:** View register as 32 raw bits rather than as a single 32-bit number
 - Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole.
- Introduce two new classes of instructions:
 - Logical & Shift Ops



CS61C L9 MIPS Proc II, Logical & Shift Ops, and Instruction Representation I (12)

Chae, Summer 2008 ©UCB

Logical Operators (1/3)

- Two basic logical operators:
 - AND: outputs 1 only if **both** inputs are 1
 - OR: outputs 1 if **at least one** input is 1
- Truth Table: standard table listing all possible combinations of inputs and resultant output for each. E.g.,

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1



Logical Operators (2/3)

- Logical Instruction Syntax:
 - 1 2,3,4
- where
 - 1) operation name
 - 2) register that will receive value
 - 3) first operand (register)
 - 4) second operand (register) or immediate (numerical constant)
- In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
- Again, rigid syntax, simpler hardware



Logical Operators (3/3)

- Instruction Names:
 - and, or: Both of these expect the third argument to be a register
 - andi, ori: Both of these expect the third argument to be an immediate
- MIPS Logical Operators are all **bitwise**, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.
 - C: Bitwise AND is & (e.g., $z = x \& y;$)
 - C: Bitwise OR is | (e.g., $z = x | y;$)



Uses for Logical Operators (1/3)

- Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.
- This can be used to create a **mask**.
 - Example:

```
1011 0110 1010 0100 0011 1101 1001 1010
mask: 0000 0000 0000 0000 0000 1111 1111 1111
```
 - The result of anding these:

```
0000 0000 0000 0000 0000 1101 1001 1010
```



Uses for Logical Operators (2/3)

- The second bitstring in the example is called a **mask**. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).
- Thus, the and operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
 - In particular, if the first bitstring in the above example were in \$t0, then the following instruction would mask it:

```
andi $t0, $t0, 0xFFFF
```



Uses for Logical Operators (3/3)

- Similarly, note that oring a bit with 1 produces a 1 at the output while oring a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
 - For example, if \$t0 contains 0x12345678, then after this instruction:

```
ori $t0, $t0, 0xFFFF
```
 - ... \$t0 contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).



Shift Instructions (1/5)

- Move (shift) all the bits in a word to the left or right by a number of bits.

- Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift left by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000



Shift Instructions (2/5)

- Shift Instruction Syntax:

1 2,3,4

- where

- 1) operation name
- 2) register that will receive value
- 3) first operand (register)
- 4) shift amount (constant < 32)

- MIPS shift instructions:

1. **sll** (shift left logical): shifts left and **fills emptied bits with 0s**
2. **srl** (shift right logical): shifts right and **fills emptied bits with 0s**
3. **sra** (shift right arithmetic): shifts right and **fills emptied bits by sign extending**



Shift Instructions (3/5)

- MIPS shift instructions with variable values:

sllv, srlv, sra

Just like their immediate counterparts, except these shift by an amount specified in a register instead of an immediate



Shift Instructions (4/5)

- Example: shift right arith by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift right arith by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110



Shift Instructions (5/5)

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

a *= 8; (in C)

would compile to:

sll \$s0, \$s0, 3 (in MIPS)

- Likewise, shift right to divide by powers of 2

- remember to use sra



Peer Instruction

```
r: ... # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
...   ### PUSH REGISTER(S) TO STACK?
jal e # Call e
...   # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
jr $ra # Return to caller of r
```

```
e: ... # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
jr $ra # Return to r
```

What does r have to push on the stack before "jal e"?

- 1: 1 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 2: 2 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 3: 3 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 4: 4 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 5: 5 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 6: 6 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 7: 0 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)



Administrivia

- Midterm
 - Monday 7/21, 7-10pm in 155 Dwinelle
 - Contact me immediately if you have a conflict
- “Faux” Midterm?
- Midterm Review Session?

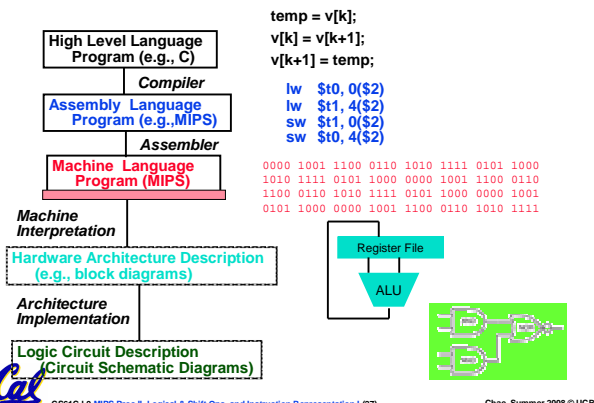


Administrivia

- Assignments
 - HW2 due 7/7 TODAY @ 11:59pm
 - Quiz 4 due 7/8 @ 11:59pm
 - Proj1 due 7/11 @ 11:59pm



61C Levels of Representation (abstractions)



Overview – Instruction Representation

- Big idea: stored program
 - consequences of stored program
- Instructions as numbers
- Instruction encoding
- MIPS instruction format for Add instructions



Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
 - 1) Instructions are represented as numbers.
 - 2) Therefore, entire programs can be stored in memory to be read or written just like numbers (data).
- Simplifies SW/HW of computer systems:
 - Memory technology for data also used for programs



Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
 - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
 - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: “Program Counter” (PC)
 - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name



Consequence #2: Binary Compatibility

- Programs are distributed in binary form
 - Programs bound to specific instruction set
 - Different version for **Macintoshes** and **PCs**
- New machines want to run old programs (“binaries”) as well as programs compiled to new instructions
- Leads to “backwards compatible” instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today



Instructions as Numbers (1/2)

- Currently all data we work with is in words (32-bit blocks):
 - Each register is a word.
 - `lw` and `sw` both access memory one word at a time.
- So how do we represent instructions?
 - Remember: Computer only understands 1s and 0s, so “`add $t0,$0,$0`” is meaningless.
 - MIPS wants simplicity: since data is in words, make instructions be words too



Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into “**fields**”.
- Each field tells computer something about instruction.
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
 - R-format
 - I-format
 - J-format



Instruction Formats

- **I-format**: used for instructions with immediates, `lw` and `sw` (since the offset counts as an immediate), and the branches (`beq` and `bne`),
 - (but not the shift instructions; later)
- **J-format**: used for `j` and `jal`
- **R-format**: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way.



R-Format Instructions (1/5)

- Define “**fields**” of the following number of bits each: $6 + 5 + 5 + 5 + 5 + 6 = 32$

6	5	5	5	5	6
---	---	---	---	---	---

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

- **Important**: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.
 - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.



R-Format Instructions (2/5)

- What do these field integer values tell us?
 - **opcode**: partially specifies what instruction it is
 - Note: This number is equal to 0 for all R-Format instructions.
 - **funct**: combined with `opcode`, this number exactly specifies the instruction
 - Question: Why aren't `opcode` and `funct` a single 12-bit field?
 - Answer: We'll answer this later.



R-Format Instructions (3/5)

• More fields:

- **rs** (Source Register): *generally* used to specify register containing first operand
- **rt** (Target Register): *generally* used to specify register containing second operand (note that name can be misleading)
- **rd** (Destination Register): *generally* used to specify register which will receive result of computation



R-Format Instructions (4/5)

• Notes about register fields:

- Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
- The word “generally” was used because there are exceptions that we’ll see later. E.g.,
 - `mult` and `div` have nothing important in the `rd` field since the dest registers are `hi` and `lo`
 - `mfhi` and `mflo` have nothing important in the `rs` and `rt` fields since the source is determined by the instruction (p. 264 P&H)



R-Format Instructions (5/5)

• Final field:

- **shamt**: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31).
- This field is set to 0 in all but the shift instructions.
- For a detailed description of field usage for each instruction, see green insert in COD 3/e
- (You can bring with you to all exams)



R-Format Example (1/2)

• MIPS Instruction:

`add $8, $9, $10`

`opcode = 0` (look up in table in book)

`funct = 32` (look up in table in book)

`rd = 8` (destination)

`rs = 9` (first operand)

`rt = 10` (second operand)

`shamt = 0` (not a shift)



R-Format Example (2/2)

• MIPS Instruction:

`add $8, $9, $10`

Decimal number per field representation:

0	9	10	8	0	32
---	---	----	---	---	----

Binary number per field representation:

000000	01001	01010	01000	00000	100000
--------	-------	-------	-------	-------	--------

hex representation: `012A 4020`_{hex}

decimal representation: `19,546,144`_{ten}

- Called a **Machine Language Instruction**



In conclusion...

• Logical and Shift Instructions

- Operate on individual bits (arithmetic operate on entire word)
- Use to isolate fields, either by masking or by shifting back & forth
- Use **shift left logical**, `sll`, for multiplication by powers of 2
- Use **shift right arithmetic**, `sra`, for division by powers of 2

• Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use `lw` and `sw`).

- Computer actually stores programs as a series of these 32-bit numbers.

• MIPS Machine Language Instruction: 32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
---	--------	----	----	----	-------	-------

