inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #10 – Instruction Representation II
2007-7-8

Albert Chae, Instructor

Review

- There are register calling conventions!
  - Saved: $s0-7, $sp
  - Volatile: $ra, $v0-1, $a0-3, $t0-9
- CalleR must save volatile registers before calling calleE
- CalleE must save saved registers before using them, and then restore them before returning
- Prologue, Epilogue

Review…

- 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

  Syntax: op rd, rs, rt

R	opcode	rs	rt	rd	shamt	funct

I-Format Instructions (1/4)

- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:
  - First notice that, if instruction has immediate, then it uses at most 2 registers.

I-Format Instructions (2/4)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 16 = 32 bits

6	5	5	16

- Again, each field has a name:

opcode	rs	rt	immediate

- Key Concept: Only one field is inconsistent with R-format. Most importantly, `opcode` is still in same location.

I-Format Instructions (3/4)

- What do these fields mean?
  - `opcode`: same as before except that, since there's no `funct` field, `opcode` uniquely specifies an instruction in I-format
  - This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent with other formats.
  - `rs`: specifies the *only* register operand (if there is one)
  - `rt`: specifies register which will receive result of computation (this is why it's called the *target* register "rt")

I-Format Instructions (4/4)

- The Immediate Field:
  - `addi`, `slti`, `sltiu`, the immediate is sign-extended to 32 bits. Thus, it's treated as a signed integer.
  - 16 bits ➔ can be used to represent immediate up to 2^{16} different values
  - This is large enough to handle the offset in a typical `lw` or `sw`, plus a vast majority of values that will be used in the `slti` instruction.
  - We'll see what to do when the number is too big in our next lecture…

I-Format Example (1/2)

- MIPS Instruction:

```
addi    $21,$22,-50
```

`opcode` = 8 (look up in table in book)

`rs` = 22 (register containing operand)

`rt` = 21 (target register)

`immediate` = -50 (by default, this is decimal)

I-Format Example (2/2)

- MIPS Instruction:

```
addi    $21,$22,-50
```

Decimal/field representation:

8	22	21	-50

Binary/field representation:

001000	10110	10101	1111111111001110

hexadecimal representation: $22D5\ FFCE_{hex}$

decimal representation: $584{,}449{,}998_{ten}$

I-Format Problems (0/3)

- Problem 0: Unsigned # sign-extended?
  - `addiu`, `sltiu`, sign-extends immediates to 32 bits. Thus, # is a "signed" integer.
- Rationale
  - `addiu` "unsigned" means can add w/out overflow
    - See K&R pp. 230, 305
    - `addiu` ← This u isn't talking about the immediate
  - `sltiu` suffers so that we can have ez HW
    - Does this mean we'll get wrong answers?
    - Nope, it means assembler has to handle any unsigned immediate $2^{15} \le n < 2^{16}$ (I.e., with a 1 in the 15th bit and 0s in the upper 2 bytes) as it does for numbers that are too large. ⇒

I-Format Problems (1/3)

- Problem 1:
  - Chances are that `addi`, `lw`, `sw` and `slti` will use immediates small enough to fit in the immediate field.
  - …but what if it's too big?
  - We need a way to deal with a 32-bit immediate in any I-format instruction.

I-Format Problems (2/3)

- Solution to Problem 1:
  - Handle it in software + new instruction
  - Don't change the current instructions: instead, add a new instruction to help out
- New instruction:

```
lui    register, immediate
```

  - stands for Load Upper Immediate
  - takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register
  - sets lower half to 0s

I-Format Problems (3/3)

- Solution to Problem 1 (continued):
  - So how does `lui` help us?
  - Example:

```
addi   $t0,$t0, 0xABABCDCD
```

becomes:

```
lui    $at, 0xABAB
ori    $at, $at, 0xCDCD
add    $t0,$t0,$at
```

  - Now each I-format instruction has only a 16-bit immediate.
  - Wouldn't it be nice if the assembler would this for us automatically? (later)

Peer Instruction

Which instruction has same representation as 35_{ten}?

1. add $0, $0, $0	opcode	rs	rt	rd	shamt	funct
2. subu $s0,$s0,$s0	opcode	rs	rt	rd	shamt	funct
3. lw $0, 0($0)	opcode	rs	rt	offset		
4. addi $0, $0, 35	opcode	rs	rt	immediate		
5. subu $0, $0, $0	opcode	rs	rt	rd	shamt	funct

6. Trick question!
   Instructions are not numbers

Registers numbers and names:
0: $0, .. 8: $t0, 9:$t1, ..15: $t7, 16: $s0, 17: $s1, .. 23: $s7

Opcodes and function fields (if necessary)

- `add`: opcode = 0, funct = 32
- `subu`: opcode = 0, funct = 35
- `addi`: opcode = 8
- `lw`: opcode = 35

Peer Instruction Answer

Which instruction bit pattern = number 35?

1. add $0, $0, $0	0	0	0	0	0	32
2. subu $s0,$s0,$s0	0	16	16	16	0	35
3. lw $0, 0($0)	35	0	0			0
4. addi $0, $0, 35	8	0	0			35
5. subu $0, $0, $0	0	0	0	0	0	35

6. Trick question!
   Instructions != numbers

Registers numbers and names:
0: $0, …, 8: $t0, 9:$t1, …,16: $s0, 17: $s1, …,

Opcodes and function fields

- `add`: opcode = 0, function field = 32
- `subu`: opcode = 0, function field = 35
- `addi`: opcode = 8
- `lw`: opcode = 35

Administrivia

- Assignments
  - Quiz 4 due 7/8 @ 11:59pm
  - Proj1 due 7/11 @ 11:59pm
  - HW3 due 7/15 @ 11:59pm
  - Proj2 due 7/18 @ 11:59pm
- Midterm
  - Monday 7/21, 7-10pm in 155 Dwinelle
  - Faux midterm and review session TBA
- Archived Webcasts
  - wla.berkeley.edu or link on webpage to sp08

Branches: PC-Relative Addressing (1/5)

- Use I-Format

opcode	rs	rt	immediate

- `opcode` specifies `beq` v. `bne`
- `rs` and `rt` specify registers to compare
- What can `immediate` specify?
  - `Immediate` is only 16 bits
  - PC (Program Counter) has byte address of current instruction being executed; 32-bit pointer to memory
  - So `immediate` cannot specify entire address to branch to.

Branches: PC-Relative Addressing (2/5)

- How do we usually use branches?
  - Answer: `if-else`, `while`, `for`
  - Loops are generally small: typically up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (`j` and `jal`), not the branches.
- Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount

Branches: PC-Relative Addressing (3/5)

- Solution to branches in a 32-bit instruction: PC-Relative Addressing
- Let the 16-bit `immediate` field be a signed two's complement integer to be *added* to the PC if we take the branch.
- Now we can branch ± 2^{15} bytes from the PC, which should be enough to cover almost any loop.
- Any ideas to further optimize this?

Branches: PC-Relative Addressing (4/5)

- Note: Instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with `00` in binary).
  - So the number of bytes to add to the PC will always be a multiple of 4.
  - So specify the `immediate` in words.
- Now, we can branch ± 2^{15} words from the PC (or ± 2^{17} bytes), so we can handle loops 4 times as large.

Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
  - If we don't take the branch:

    PC = PC + 4

    PC+4 = byte address of next instruction
  - If we do take the branch:

    PC = (PC + 4) + (`immediate` * 4)
  - Observations
    - `Immediate` field specifies the number of words to jump, which is simply the number of instructions to jump.
    - `Immediate` field can be positive or negative.
    - Due to hardware, add `immediate` to (PC+4), not to PC; will be clearer why later in course

Branch Example (1/3)

- MIPS Code:

```
Loop:  beq   $9,$0,End
       add   $8,$8,$10
       addi  $9,$9,-1
       j     Loop
End:
```

- `beq` branch is I-Format:

  `opcode` = 4 (look up in table)

  `rs` = 9 (first operand)

  `rt` = 0 (second operand)

  `immediate` = ???

Branch Example (2/3)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addi  $8,$8,$10
      addi  $9,$9,-1
      j     Loop
End:
```

- `Immediate` Field:
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch.
  - In `beq` case, `immediate` = 3

Branch Example (3/3)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addi  $8,$8,$10
      addi  $9,$9,-1
      j     Loop
End:
```

decimal representation:

4	9	0	3

binary representation:

000100	01001	00000	0000000000000011

Questions on PC-addressing

- Does the value in branch field change if we move the code?
- What do we do if destination is > 2^{15} instructions away from branch?
- Why do we need all these addressing modes? Why not just one?

Green Sheet Errors

- Section 1: The Core Instruction Set
  - lb, lbu, lw scratch out 0/
  - sll, srl shift rt not rs so change R[rs] to R[rt]
  - jal should be R[31] = PC + 4, not +8
- Section 2: Register Name, Number, Use, Call Convention
  - $ra is not preserved across calls so make yes a no

J-Format Instructions (1/5)

- For branches, we assumed that we won't want to branch too far, so we can specify *change* in PC.
- For general jumps (`j` and `jal`), we may jump to *anywhere* in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.

J-Format Instructions (2/5)

- Define "fields" of the following number of bits each:

6 bits	26 bits

- As usual, each field has a name:

opcode	target address

- Key Concepts
  - Keep `opcode` field identical to R-format and I-format for consistency.
  - Combine all other fields to make room for large target address.

J-Format Instructions (3/5)

- For now, we can specify 26 bits of the 32-bit bit address.
- Optimization:
  - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always 00 (in binary).
  - So let's just take this for granted and not even specify them.

J-Format Instructions (4/5)

- Now specify 28 bits of a 32-bit address
- Where do we get the other 4 bits?
  - By definition, take the 4 highest order bits from the PC.
  - Technically, this means that we cannot jump to *anywhere* in memory, but it's adequate 99.9999…% of the time, since programs aren't that long
    - only if straddle a 256 MB boundary
  - If we absolutely need to specify a 32-bit address, we can always put it in a register and use the `jr` instruction.

J-Format Instructions (5/5)

- Summary:
  - New PC = { PC[31..28], target address, 00 }
- Understand where each part came from!
- Note: { , , } means concatenation
  { 4 bits , 26 bits , 2 bits } = 32 bit address
  - { 1010, 11111111111111111111111111, 00 }
    = 10101111111111111111111111111100
  - Note: Book uses ||

Peer Instruction Question

(for A,B) When combining two C files into one executable, recall we can compile them independently & then merge them together.

A. Jump insts don't require any changes.
B. Branch insts don't require any changes.
C. You now have all the tools to be able to "decompile" a stream of 1s and 0s into C!

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

In conclusion…

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

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Branches use PC-relative addressing, Jumps use absolute addressing.
- Disassembly is simple and starts by decoding `opcode` field. (more in a week)