EECE 321: Computer Organization

Mohammad M. Mansour
Dept. of Electrical and Compute Engineering
American University of Beirut

Lecture 10: MIPS Instruction Formats

Announcements

- Install and learn the SPIM tool (from textbook CD)
  - MIPS assembler simulator
- Assignments:
  - HW3 due Monday Mar. 15 @ 5:00pm
  - MP1 due Friday March 12 @ 5:00pm
    - Submit on Moodle (one submission per team)
- Makeup lectures on the following dates (SPIM, Modelsim, problem session):
  - M: Mar. 15
  - F: Mar. 19
  - M: Mar. 22
- Makeup lectures on Saturday March 13
  - Time and place TBA

Instructions as Numbers

- All data and instructions are stored in memory as numbers, and hence must have addresses.
- 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?
  - MIPS wants simplicity: since data is in words, make instructions be (32-bit) words too
- 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 regularity and simplicity, so define 3 basic types of instruction formats:
    - I-format
    - J-format
    - R-format
- 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)
- J-format: used for j and jal
- R-format: used for all other instructions

R-Format Instructions

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

opcode	rs	rt	rd	shamt	funct
6b	5b	5b	5b	5b	6b

- 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 variant of the instruction.
  - 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 is misleading)
  - rd (Destination Register): generally used to specify register which will receive result of computation
  - 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.
- See green cover of P&H textbook for a detailed description of field usage for each instruction.
  - Appendix B.10 lists all MIPS R2000 Assembly instructions and their format.

R-Format Example

opcode	rs	rt	rd	shamt	funct
6b	5b	5b	5b	5b	6b

- MIPS Instruction: add $8, $9, $10
  - opcode = 0 (look up in table in book)
  - funct = 32 (look up in table in book)
  - rs = 9 (first operand)
  - rt = 10 (second operand)
  - rd = 8 (destination)
  - shamt = 0 (not a shift)
- Decimal number per field representation:

opcode	rs	rt	rd	shamt	funct
0	9	10	8	0	32

- Binary number per field representation:

opcode	rs	rt	rd	shamt	funct
000000	01001	01010	01000	00000	100000

- hex representation: 0x 012A 4020
- Decimal representation: 19,546,144$_{(10)}$
- Called Machine Language Instruction

Name	Register Number
$zero	0
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$gp	28
$sp	29
$fp	30
$ra	31

I-Format Instructions

- What about instructions with immediates?
  - 5-bit field only represents numbers up to 31: immediates may be much larger than this
  - Ideally, MIPS would have only 1 inst. format: 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.
- Define "fields" with the following number of bits each: 6 + 5 + 5 + 16 = 32 bits

opcode	rs	rt	funct
6b	5b	5b	16b

- Key Concept: Only one field is inconsistent with R-format. Most importantly, opcode is still in same location.
  - opcode: same as before except that, since there's no funct field, opcode uniquely specifies an instruction in I-format. opcode is either 1, or 4-31.
  - 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
  - Immediate: 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 immediates up to 2^{16} different values

I-Format Example

opcode	rs	rt	immediate
6	5	5	16

- 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)
- Decimal/field representation:

opcode	rs	rt	immediate
8	22	21	-50

- Binary/field representation:

opcode	rs	rt	immediate
001000	10110	10101	1111 1111 1100 1110

- hexadecimal representation: 0x22D5 FFCE
- decimal representation: $584{,}449{,}998_{(10)}$.

Name	Register Number
$zero	0
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$gp	28
$sp	29
$fp	30
$ra	31

Another Example

- Which instruction has same representation as $35_{(10)}$?
  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. Not possible

- 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

I-Format Limitations

- Chances are that addi, lw, sw and slti will use immediates small enough to fit in the immediate field.
  - We need a way to deal with a 32-bit immediate in any I-format instruction.
- Solution: 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
- So how does lui help us?
- Example: addi $t0, $t0, 0xABABCDCD
  - The immediate is too big to fit in a 16-bit field
  - So need to make use of lui instruction

    ```
    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 do this for us automatically? (later)

Branches: PC-Relative Addressing

opcode	rs	rt	immediate
6b	5b	5b	16b

- Use I-Format:
  - 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.
- 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
- 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 (actually added to PC+4).

Branches: PC-Relative Addressing

- Now we can branch ± 2^{15} bytes from the PC, which should be enough to cover almost any loop.
- 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.
- Branch Calculation:
  - If we don't take the branch:
    - PC $\leftarrow$ PC + 4
    - PC+4 is the byte address of next instruction
  - If we do take the branch:
    - PC $\leftarrow$ (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

Branch Example

opcode	rs	rt	immediate
6b	5b	5b	16b

- 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 = ???
- 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

decimal

opcode	rs	rt	immediate
4	9	0	3

binary

opcode	rs	rt	immediate
000100	01001	00000	0000 0000 0000 0011

Questions on PC-Relative 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?
- Since it's limited to ± 2^{15} instructions, doesn't this generate lots of extra MIPS instructions?
- Why do we need all these addressing modes? Why not just one?