CS 61C: Great Ideas in Computer Architecture

MIPS Instruction Representation II

Dan Garcia

1

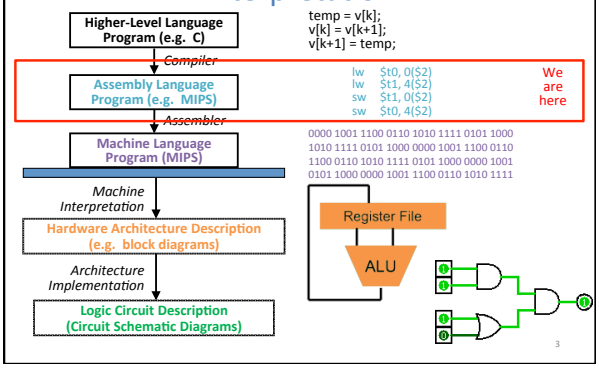
Review of Last Lecture

- **Simplifying MIPS:** Define instructions to be same size as data word (one word) so that they can use the same memory
  - 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
I:	opcode	rs	rt	immediate		

2

Great Idea #1: Levels of Representation/ Interpretation



3

I-Format Immediates

- `immediate` (16): *two's complement* number
  - All computations done in words, so 16-bit immediate must be *extended* to 32 bits
  - Green Sheet specifies ZeroExtImm or SignExtImm based on instruction
- Can represent 2^{16} different immediates
  - This is large enough to handle the offset in a typical `lw`/`sw`, plus the vast majority of values for `slti`

5

Dealing With Large Immediates

- How do we deal with 32-bit immediates?
  - Sometimes want to use immediates > ± 2^{15} with `addi`, `lw`, `sw` and `slti`
  - Bitwise logic operations with 32-bit immediates
- **Solution:** Don't mess with instruction formats, just add a new instruction
- Load Upper Immediate (`lui`)
  - `lui reg,imm`
  - Moves 16-bit `imm` into upper half (bits 16-31) of `reg` and zeros the lower half (bits 0-15)

6

`lui` Example

- Want: `addiu $t0,$t0,0xABABCDCD`
  - This is a pseudo-instruction!
- Translates into:

```
lui  $at,0xABAB      # upper 16
ori  $at,$at,0xCDCD  # lower 16
addu $t0,$t0,$at     # move
```

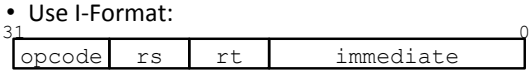
Only the assembler gets to use $at

- Now we can handle everything with a 16-bit immediate!

7

Branching Instructions

- `beq` and `bne`
  - Need to specify an address to go to
  - Also take two registers to compare
- Use I-Format:



  - `opcode` specifies `beq` (4) vs. `bne` (5)
  - `rs` and `rt` specify registers
  - How to best use `immediate` to specify addresses?

8

Branching Instruction Usage

- Branches typically used for loops (`if-else`, `while`, `for`)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

9

PC-Relative Addressing

- PC-Relative Addressing: Use the `immediate` field as a two's complement offset to PC
  - Branches generally change the PC by a small amount
  - Can specify ± 2^{15} addresses from the PC
- So just how much of memory can we reach?

10

Branching Reach

- **Recall:** MIPS uses 32-bit addresses
  - Memory is byte-addressed
- Instructions are *word-aligned*
  - Address is always multiple of 4 (in bytes), meaning it ends with `0b00` in binary
  - Number of bytes to add to the PC will always be a multiple of 4
- Immediate specifies words instead of bytes
  - Can now branch ± 2^{15} words
  - We can reach 2^{16} instructions = 2^{18} bytes around PC

11

Branch Calculation

- If we don't take the branch:
  - `PC = PC + 4` = next instruction
- If we do take the branch:
  - `PC = (PC+4) + (immediate*4)`
- **Observations:**
  - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)
  - Branch from `PC+4` for hardware reasons; will be clear why later in the course

12

Branch Example (1/2)

- MIPS Code:

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

Start counting from instruction AFTER the branch

1
2
3

- I-Format fields:
  - `opcode` = 4 (look up on Green Sheet)
  - `rs` = 9 (first operand)
  - `rt` = 0 (second operand)
  - `immediate` = 3

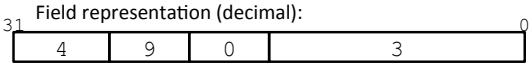
13

Branch Example (2/2)

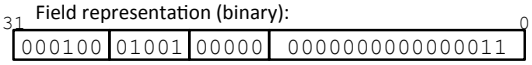
- MIPS Code:

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

Field representation (decimal):



Field representation (binary):



14

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no
- What do we do if destination is > 2^{15} instructions away from branch?
  - Other instructions save us
  - `beq $s0,$0,far` `# next instr` → `bne $s0,$0,next` `j far` `next: # next instr`

15

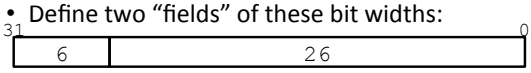
J-Format Instructions (1/4)

- For branches, we assumed that we won't want to branch too far, so we can specify a *change* in the PC
- For general jumps (`j` and `jal`), we may jump to *anywhere* in memory
  - Ideally, we would 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

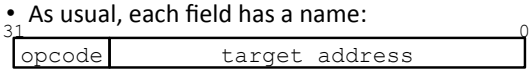
19

J-Format Instructions (2/4)

- Define two "fields" of these bit widths:



- As usual, each field has a name:



- **Key Concepts:**
  - Keep `opcode` field identical to R-Format and I-Format for consistency
  - Collapse all other fields to make room for large target address

20

J-Format Instructions (3/4)

- We can specify 2^{26} addresses
  - Still going to word-aligned instructions, so add `0b00` as last two bits (multiply by 4)
  - This brings us to 28 bits of a 32-bit address
- Take the 4 highest order bits from the PC
  - Cannot reach *everywhere*, but adequate almost all of the time, since programs aren't that long
  - Only problematic if code straddles a 256MB boundary
- If necessary, use 2 jumps or `jr` (R-Format) instead

21

J-Format Instructions (4/4)

- Jump instruction:
  - New PC = { (PC+4)[31..28], target address, 00 }
- Notes:
  - { , , } means concatenation
    { 4 bits , 26 bits , 2 bits } = 32 bit address
    - Book uses || instead
  - Array indexing: [31..28] means highest 4 bits
  - For hardware reasons, use PC+4 instead of PC

22

Question: When combining two C files into one executable, we can compile them independently and then merge them together.

When merging two or more binaries:

1) Jump instructions don't require any changes
2) Branch instructions don't require any changes

	1	2
a)	F	F
b)	F	T
c)	T	F
d)	T	T

23

Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
  - e.g. assignment (`a=b`) via addition with `0`
- MIPS has a set of "pseudo-instructions" to make programming easier
  - More intuitive to read, but get translated into actual instructions later
- Example:

```
move dst,src translated into
addi dst,src,0
```

25

Assembler Pseudo-Instructions

- List of pseudo-instructions: http://en.wikipedia.org/wiki/MIPS_architecture#Pseudo_instructions
  - List also includes instruction translation
- Load Address (`la`)
  - `la dst,label`
  - Loads address of specified label into `dst`
- Load Immediate (`li`)
  - `li dst,imm`
  - Loads 32-bit immediate into `dst`
- MARS has additional pseudo-instructions
  - See Help (F1) for full list

26

Assembler Register

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses a regular register, it'll overwrite whatever the program has put into it
- Solution:
  - Reserve a register ($1 or $at for "assembler temporary") that assembler will use to break up pseudo-instructions
  - Since the assembler may use this at any time, it's not safe to code with it

27

MAL vs. TAL

- True Assembly Language (TAL)
  - The instructions a computer understands and executes
- MIPS Assembly Language (MAL)
  - Instructions the assembly programmer can use (includes pseudo-instructions)
  - Each MAL instruction becomes 1 or more TAL instruction
- TAL ⊂ MAL

28

Summary

- I-Format: instructions with immediates, `lw`/`sw` (offset is immediate), and `beq`/`bne`
  - But not the shift instructions
  - Branches use PC-relative addressing

I:	opcode	rs	rt	immediate

- J-Format: `j` and `jal` (but not `jr`)
  - Jumps use absolute addressing

J:	opcode	target address

- R-Format: all other instructions

R:	opcode	rs	rt	rd	shamt	funct

29