CSE 31
Computer Organization

Lecture 13 – Instruction Format (2)

Announcement

- Project #1
  - Due at 11:59pm on 3/22, Friday (no more late submission)
  - You must demo your submission to your TA during week of 4/1, in lab.
- Lab #7 this week
  - Due after Spring Break
  - Demo your lab within 7 days after due dates
- HW #4 in CatCourses
  - Due Monday (4/1) at 11:59pm
- Reading assignment
  - Chapter 1.6, 6.1-6.3 of zyBooks
    - Make sure to do the Participation Activities
    - Due Wednesday (4/3) at 11:59pm

Levels of Representation (abstractions)

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g.,MIPS)

```
lw   $t0, 0($s2)
lw   $t1, 4($s2)
sw   $t1, 0($s2)
sw   $t0, 4($s2)
```

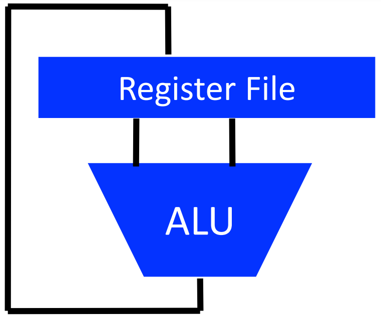
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

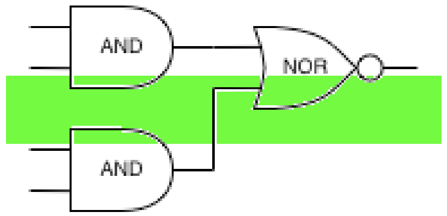
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Instruction Formats

- **3 formats**
- I-format: used for instructions with immediates, `lw` and `sw` (since offset counts as an immediate), and branches (`beq` and `bne`),
  - (but not the shift instructions; later)
- J-format: used for `j` and `jal`
- R-format: used for all other instructions
- Why 3 different formats?
  - 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?
  - 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 is misleading)
  - `rd` (Destination Register): generally used to specify register which will **receive result** of computation

R-Format Instructions (4/5)

- Notes about register fields:
  - Do we need more than 5 bits?
  - 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
    - **`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.

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 MIPS_Reference_Sheet.pdf in CatCourses (You will be provided in all exams)

R-Format Example (1/2)

- MIPS Instruction:

```
add    $t0,$t1,$t2
add    $8,$9,$10
```

`opcode` = 0 (look up in table)

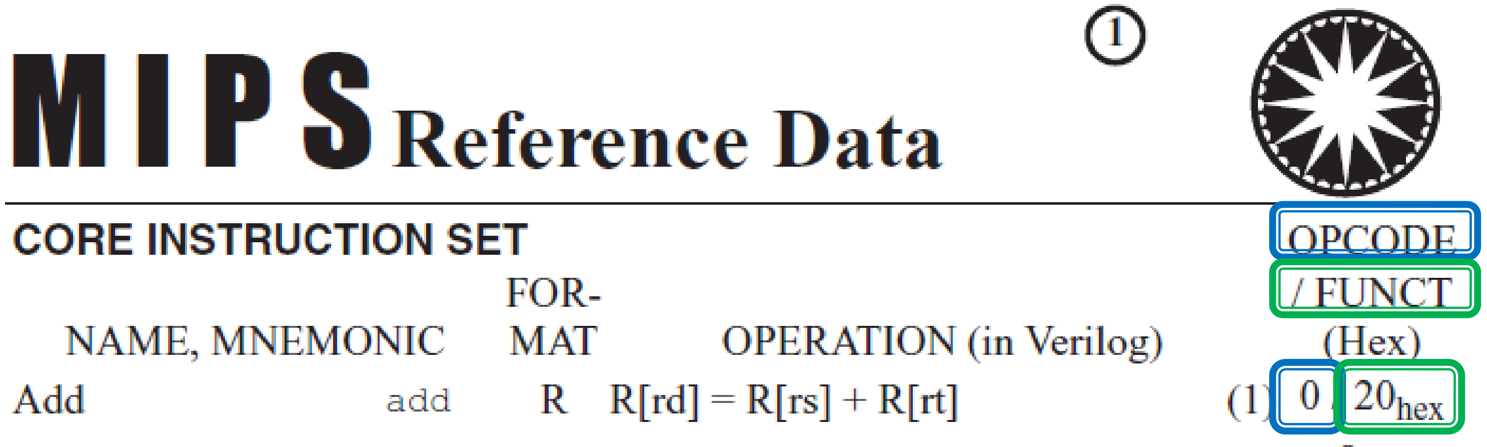
`funct` = 32 (look up in table)

`rd` = 8 (destination)

`rs` = 9 (first operand)

`rt` = 10 (second operand)

`shamt` = 0 (not a shift)



MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}

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

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

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

Called a Machine Language Instruction

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 as possible with other formats while leaving as much space as possible for immediate field.
  - **`rs`**: specifies a 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`") or other operand for some instructions.

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 later

I-Format Example (1/2)

- MIPS Instruction:

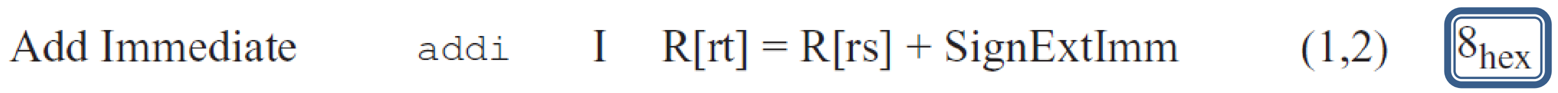
  ```
  addi    $s5,$s6,-50
  addi    $21,$22,-50
  ```

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

  `rs` = 22 (register containing operand)

  `rt` = 21 (target register)

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



Add Immediate	`addi`	I	R[rt] = R[rs] + SignExtImm	(1,2)	$\boxed{8_{hex}}$

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$_{ten}$

Quiz

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

a) add \$0, \$0, \$0
b) subu \$s0,\$s0,\$s0
c) lw \$0, 0(\$0)
d) addi \$0, \$0, 35
e) subu \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	offset		
opcode	rs	rt	immediate		
opcode	rs	rt	rd	shamt	funct

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

Quiz

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

a) add \$0, \$0, \$0
b) subu \$s0,\$s0,\$s0
c) lw \$0, 0(\$0)
d) addi \$0, \$0, 35
e) subu \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	offset		
opcode	rs	rt	immediate		
opcode	rs	rt	rd	shamt	funct

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 Problems (0/3)

- Problem 0: Unsigned # sign-extended?
  - `addiu`, `sltiu`, sign-extends immediates to 32 bits. Thus, # is a "signed" integer.
- Rationale
  - `addiu` so that can add w/out overflow
    - See K&R pp. 230, 305
  - `sltiu` suffers so that we can have easy HW
    - Does this mean we'll get wrong answers?
    - Nope, it means assembler has to handle any unsigned immediate $2^{15} \leq n < 2^{16}$ (i.e., with a `1` in the 15th bit and `0`s in the upper 2 bytes) as it does for numbers that are too large.

I-Format Problem (1/3)

- Problem:
  - 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?
    - addi $s0, $s1, 40000
    - We need a way to deal with a 32-bit immediate in any I-format instruction.

I-Format Problem (2/3)

- Solution to Problem:
  - 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 register
  - sets lower half to `0`s

I-Format Problems (3/3)

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

```
        addiu $t0,$t0, 0xABABCDCD
```

…becomes

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

R-format!

  - 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 (1/5)

- Use I-Format

opcode	rs	rt	immediate

- `opcode` specifies `beq` versus `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 typically use branches?
  - Answer: `if-else`, `while`, `for`
  - Loops are generally small: usually 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 = 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
       addu   $8,$8,$10
       addiu  $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` = ??? (Do we need to know the address of End?)

Branch Example (2/3)

- MIPS Code:

```
Loop: beq   $9,$0,End  ← PC
      addu  $8,$8,$10  ← PC + 4
      addiu $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
       addu   $8,$8,$10
       addiu  $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 different addressing modes (different ways of describing a memory address)? Why not just one?

Summary

- 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
I	opcode	rs	rt	immediate		