Instruction Representation 2

Review

- MIPS defines instructions to be same size as data (one word) so that they can use the same memory (can use `lw` and `sw`).

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

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		

- Computer actually stores programs as a series of these machine intructions.

Outline

- Branch instruction encoding
- Jump instructions
- Disassembly
- Pseudoinstructions and "True" Assembly Language (TAL) v. "MIPS" Assembly Language (MAL)

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 is 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: Though we may want to branch to anywhere in memory, a single branch will generally change the **PC** by a very small amount.

Branches: PC-Relative Addressing (3/5)

- Solution: 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 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:
```


- **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 its > 2^15 instructions?
- Since its 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?

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)

- So, we can specify 28 bits of the 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.
  - 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 (26 bits)
    || 00
  - **Note: || means concatenation**
    4 bits || 26 bits || 2 bits = 32-bit address
- **Understand where each part came from!**

Outline

- Branch instruction encoding
- Jump instructions
- Disassembly
- Pseudoinstructions and "True" Assembly Language (TAL) v. "MIPS" Assembly Language (MAL)

Decoding Machine Language

- How do we convert 1s and 0s to C code?

  Machine language => C

- For each 32 bits:
  - Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.
  - Use instruction type to determine which fields exist.
  - Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
  - Logically convert this MIPS code into valid C code. Always possible? Unique?

Decoding Example (1/7)

- Here are six machine language instructions in hex:

```
00001025
0005402A
11000003
00441020
20A5FFFF
08100001
```


- Let the first instruction be at address $4{,}194{,}304_{10}$ (0x00400000).
- Next step: convert to binary

Decoding Example (2/7)

- The six machine language instructions in binary:

```
00000000000000000001000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000010001000001000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

- Next step: identify opcode and format

Decoding Example (3/7)

- Select the opcode (first 6 bits) to determine the format:

Format:

R	00000000000000000001000000100101
R	00000000000001010100000000101010
I	00010001000000000000000000000011
R	00000000010001000001000000100000
I	00100000101001011111111111111111
J	00001000000100000000000000000001

- Look at `opcode`:
  0 means R-Format,
  2 or 3 mean J-Format,
  otherwise I-Format.

- Next step: separation of fields

Decoding Example (4/7)

- Fields separated based on format/opcode:

Format:

R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0	+3		
R	0	2	4	2	0	32
I	8	5	5	-1		
J	2	1,048,577				

- Next step: translate ("disassemble") to MIPS assembly instructions

Decoding Example (5/7)

- MIPS Assembly (Part 1):

```
0x00400000  or    $2,$0,$0
0x00400004  slt   $8,$0,$5
0x00400008  beq   $8,$0,3
0x0040000c  add   $2,$2,$4
0x00400010  addi  $5,$5,-1
0x00400014  j     0x100001
```


- Better solution: translate to more meaningful instructions (fix the branch/jump and add labels)

Decoding Example (6/7)

- **MIPS Assembly (Part 2):**

```
        or    $v0,$0,$0
Loop:   slt   $t0,$0,$a1
        beq   $t0,$0,Exit
        add   $v0,$v0,$a0
        addi  $a1,$a1,-1
  j   Loop
Exit:
```


- **Next step: translate to C code (be creative!)**

Decoding Example (7/7)

- C code:
  - Mapping: $v0: product
    $a0: multiplicand
    $a1: multiplier


```
product = 0;
while (multiplier > 0) {
    product += multiplicand;
    multiplier -= 1;
}
```

Outline

- Branch instruction encoding
- Jump instructions
- Disassembly
- Pseudoinstructions and "True" Assembly Language (TAL) v. "MIPS" Assembly Language (MAL)

Review from Last Lecture: lui

- 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?
  - If number too big, then just automatically replace `addi` with `lui`, `ori`, `add`

True Assembly Language

- Pseudoinstruction: A MIPS instruction that doesn't turn directly into a machine language instruction.
- What happens with pseudoinstructions?
  - They're broken up by the assembler into several "real" MIPS instructions.
  - But what is a "real" MIPS instruction? Answer in a few slides
- First some examples

Example Pseudoinstructions

- **Register Move**

```
move   reg2,reg1
```

Expands to:

```
add    reg2,$zero,reg1
```


- **Load Immediate**


```
li     reg,value
```


If value fits in 16 bits:

```
addi   reg,$zero,value
```

else:

```
lui    reg,upper 16 bits of value
ori    reg,$zero,lower 16 bits
```

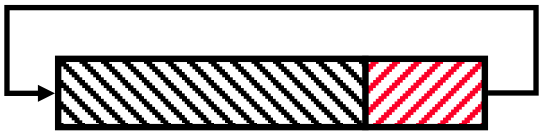
True Assembly Language

- **Problem:**
    - When breaking up a pseudoinstruction, the assembler may need to use an extra register.
    - If it uses any regular register, it'll overwrite whatever the program has put into it.

- **Solution:**
    - Reserve a register (`$1`, called `$at` for "assembler temporary") that the assembler will use when breaking up pseudo-instructions.
    - Since the assembler may use this at any time, it's not safe to code with it.

Example Pseudoinstructions

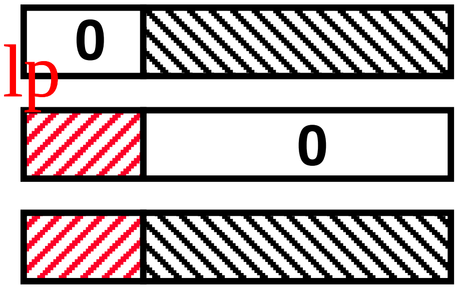
- **Rotate Right Instruction**



```
ror    reg, value
```

Expands to:

```
srl    $at, reg, value
sll    reg, reg, 32-value
or     reg, reg, $at
```



- **No operation instruction**

```
nop
```

Expands to instruction = 0_{ten},

```
sll    $0, $0, 0
```

Example Pseudoinstructions

- **Wrong operation for operand**

```
addu  reg,reg,value # should be addiu
```


If value fits in 16 bits:


```
addiu reg,reg,value
```

else:

```
lui   $at,upper 16 bits of value
ori   $at,$zero,lower 16 bits
addu  reg,reg,$at
```

True Assembly Language

- MAL (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions

- TAL (True Assembly Language): the set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)

- A program must be converted from MAL into TAL before it can be translated into 1s and 0s.

Questions on Pseudoinstructions

- How does MIPS recognize pseudoinstructions?

Peer Instruction

- Which of the codes below are pseudo-instructions (MIPS Assembly Language); that is, they are not TAL?

  i. `addi $t0, $t1, 40000`

  ii. `beq $s0, 10, Exit`

  iii. `sub $t0, $t1, 1`

A. i. only

B. ii. only

C. iii. only

D. i. and ii.

E. ii. and iii.

F. All of the above

Peer Instruction

- Which of the codes below are pseudo-instructions (MIPS Assembly Language); that is, they are not TAL?

   i. addi $t0, $t1, 40000 **40,000 > +32,767 =>`lui`,`ori`**

   ii. beq $s0, 10, Exit **beq: both must be registers**

   iii. sub $t0, $t1, 1 **sub: both must be registers; even if it was subi, there is no subi in TAL; generates addi $t0,$t1, -1**

   A. i. only

   B. ii. only

   C. iii. only

   D. i. and ii.

   E. ii. and iii.

   F. All of the above

Summary

- **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.
- Assembler expands real instruction set (TAL) with pseudoinstructions (=>MAL)

Bonus slides

- The following slides are more practice on the differences between a pointer and a value, and showing how to use pointers

Assembly Code to Implement Pointers

- dereferencing ⇒ data transfer in asm.

  `... = ... *p ...;` ⇒ load
  (get value from location pointed to by p)
  load word (lw) if int pointer,
  load byte unsigned (lbu) if char pointer

  `*p = ...;` ⇒ store
  (put value into location pointed to by p)

Assembly Code to Implement Pointers

`c` is `int`, has value 100, in memory at address 0x10000000, `p` in `$a0`, `x` in `$s0`

```
p = &c;   /* p gets 0x10000000 */
x = *p;   /* x gets 100 */
*p = 200; /* c gets 200 */
```



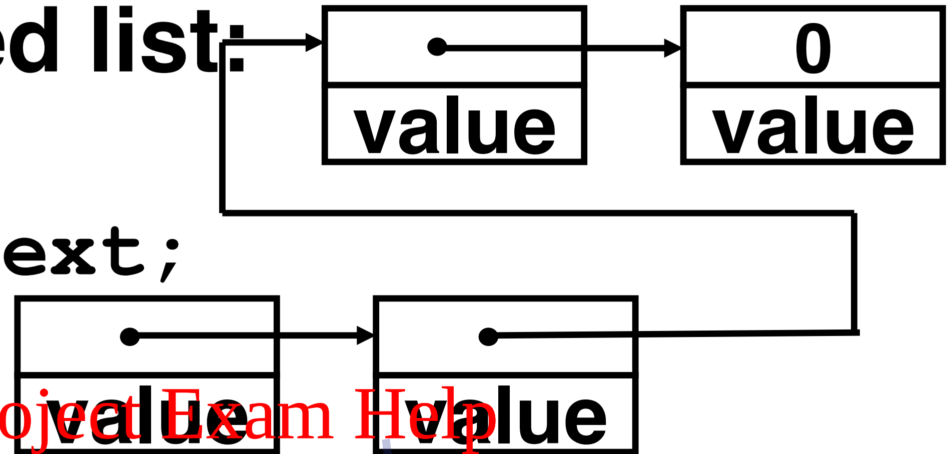
```
# p = &c;   /* p gets 0x10000000 */
lui $a0,0x1000 # p = 0x10000000

# x = *p;   /* x gets 100 */
lw   $s0, 0($a0) # dereferencing p

# *p = 200; /* c gets 200 */
addi $t0,$0,200
sw   $t0, 0($a0) # dereferencing p
```

Pointers to structures

- C Example - linked list:




```
struct node {
    struct node *next;
    int value;
};
```


If `p` is a pointer to a node, declared with `struct node *p, then:`

`(*p).value` or `p->value` for “value” field,

`(*p).next` or `p->next` for pointer to next node

Linked-list in C

```
main (void) {
   struct node *head, *temp, *ptr;
   int sum;

   /* create the nodes*/
   head = (struct node *)
                 malloc(sizeof(struct node));
   head->value = 23;
   head->next = 0;

   temp = (struct node *)
                 malloc(sizeof(struct node));
   temp->next = head;
   temp->value = 42;

   head = temp;

   /* add up the values */
   ptr = head;    sum = 0;
   while (ptr != 0) {
      sum += ptr->value;
      ptr = ptr->next;
   }
}
```

Linked-list in MIPS Assember (1/2)

```
# head:s0, temp:s1, ptr:s2, sum:s3
# create the nodes
    li   $a0,8    # sizeof(node)
    jal  malloc   # the call
    move $s0,$v0 # head gets result
    li   $t0,23
    sw   $t0,4($s0)   # head->value = 23
    sw   $zero,0($s0)# head->next = NULL

    li   $a0,8
    jal  malloc
    move $s1,$v0      # temp = malloc
    sw   $s0,0($s1)  # temp->next = head
    li   $t0,42
    sw   $t0,4($s1)  # temp->value = 42

    move $s0,$s1      # head = temp
```

Linked-list in MIPS Assember (2/2)

```
# head:s0, temp:s1, ptr:s2, sum:s3

      # add up the values
      move $s2,$s0        # ptr = head
      move $s3,$zero      # sum = 0
loop: beq  $s2,$zero exit # exit if done
      lw   $t0,4($s2)     # get value
      addu $s3,$s3,$t0    # compute new sum
      lw   $s3,0($s2)     # ptr = ptr->next
      j    loop           # repeat
exit: done
```