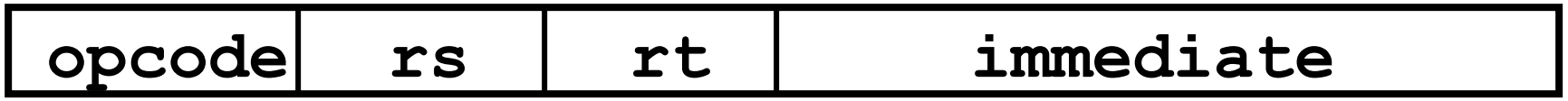
Instruction Representation 2

Outline

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

Branches: PC-Relative Addressing (1/5)



- Use I-Format
- opcode specifies **`beq`** versus **`bne`**
- **Rs** and **Rt** specify registers to compare
- What can immediate specify?
  - Immediate is only 16 bits
  - PC is 32-bit pointer to memory
  - Immediate cannot specify entire address to which we want to branch

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
  - The byte address is always a multiple of 4
  - Which means it ends with 00 in binary
  - The number of bytes to add to the PC will always be a multiple of 4
  - Thus, specify the **`immediate`** in words.
- We can branch +/- 2^{15} words from the PC (or +/- 2^{17} bytes (+/- $2^{15} * 2^{2}$ (i.e., 4 bytes))
- Thus, we can handle loops 4 times as large as a byte offset

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;
    - This will be clearer why later in course

Branch Example (1/3)

- MIPS Code:

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

- Branch is I-Format:

  opcode = 4 (look up in table)

  rs = 0 (first operand)

  rt = 9 (second operand)

  immediate = ???

Branch Example (2/3)

- MIPS Code:

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

binary representation:

000100	00000	01001	0000000000000011

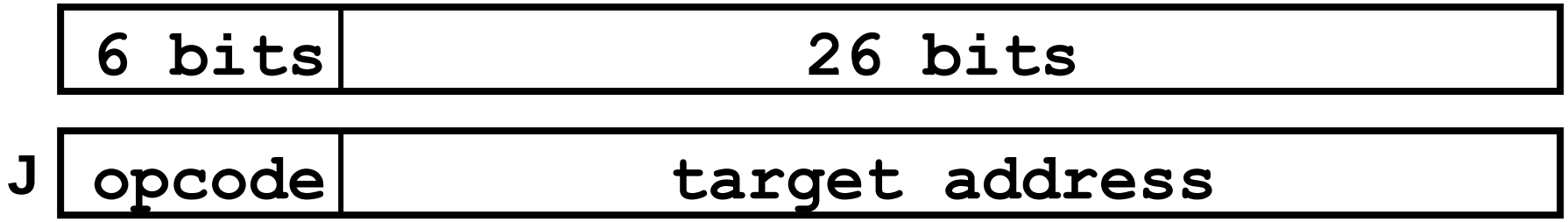
Questions on PC-addressing

- 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?

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” as above
  - As usual, each field has a name
- 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?
  - **Always take the 4 highest order bits from the PC**
  - Technically, it means that we cannot jump *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, with **||** meaning concatenation

New PC = PC[31..28] **||** target address (26 bits) **||** 00

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 assembly code?
  - Machine language → assembly
- For each 32 bits:
  - Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format
  - Instruction type determines which fields exist
  - Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number
  - MIPS code can be logically converted into HLL code such as C code. Always possible? Unique?

Decoding Example (1/7)

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

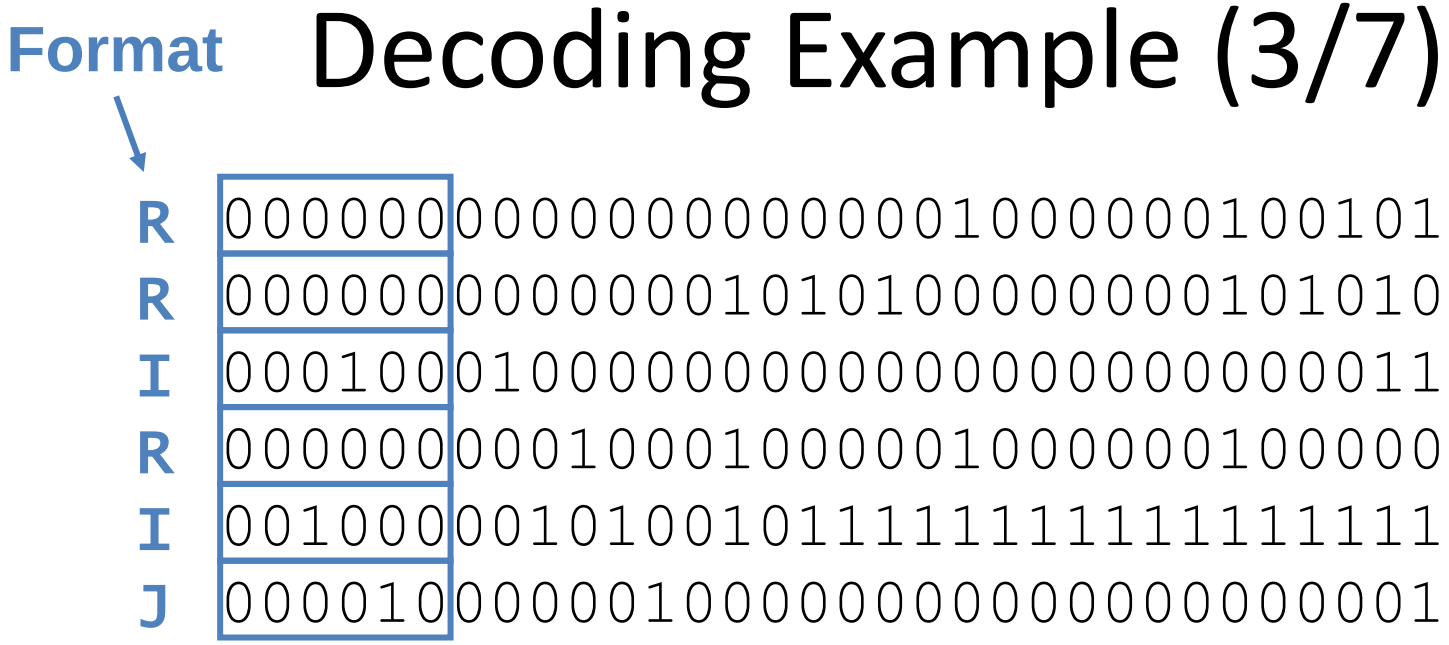
- Six machine language instructions in hex
- Let the first instruction be at address 419430410 (0x00400000).
- Next step: convert to binary

Decoding Example (2/7)

```
00000000000000000010000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000010001000001000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

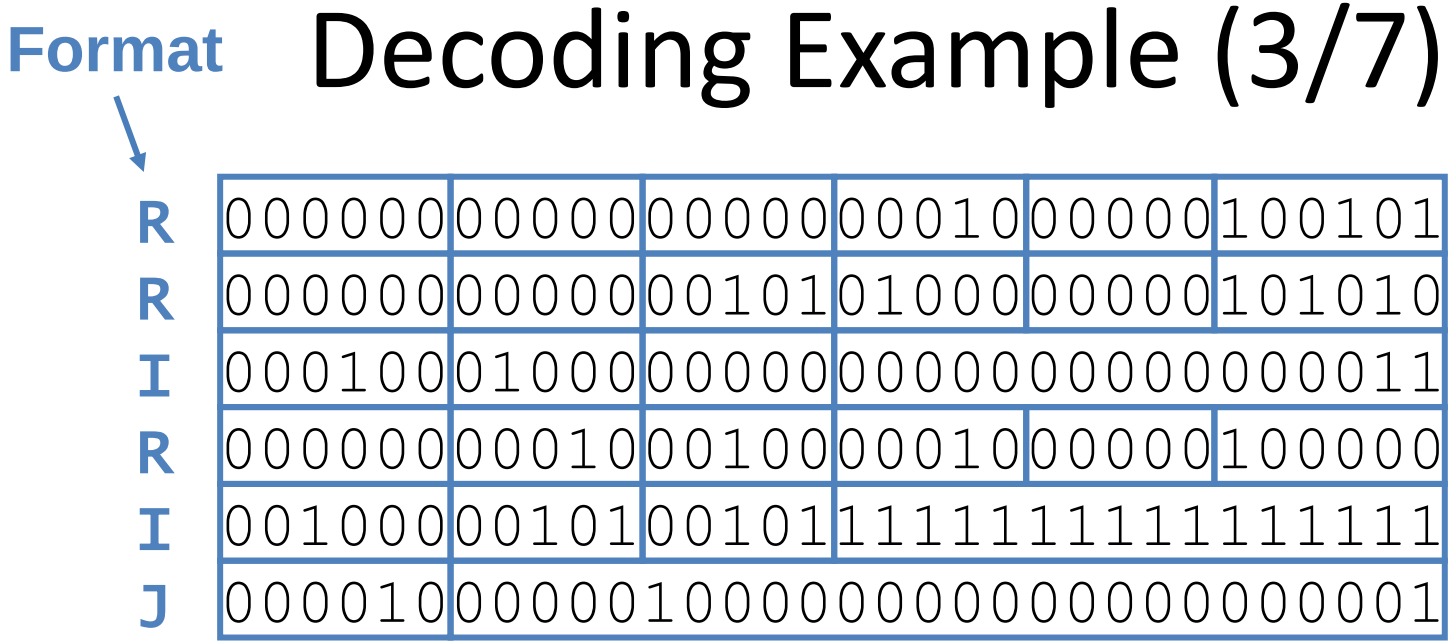
- The machine language instructions in binary
- Next step: identify opcode and format

Decoding Example (3/7)



- Opcode (first 6 bits) to determine the format
- 0 means R-Format, 2 or 3 means J-Format, otherwise I-Format
- Next step: separation of fields

Decoding Example (3/7)



- Fields separated based on format/opcode

Decoding Example (4/7)

Format							
R	0	0	0	2	0	37	or
R	0	0	5	8	0	42	slt
I	4	8	0	3			beq
R	0	2	4	2	0	32	add
I	8	5	5	-1			addi
J	2	1048577					j

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

Decoding Example (5/7)

```
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
```

- MIPS Assembly, with memory locations
- For a Better solution, translate to more meaningful instructions
  - Need to fix the branch and jump and add labels

Decoding Example (6/7)

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

- What would be a possible corresponding HLL code for this?
  - Many options

Decoding Example (7/7)

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

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

Pseudoinstructions

Outline

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

Recall Load Upper Immediate (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.
- *Assembler can do this 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, reg, 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.

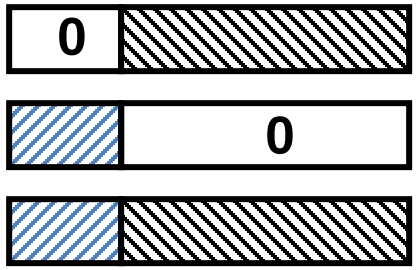
Example Pseudoinstructions

- 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, $at, lower 16 bits of value
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 assembler recognize pseudoinstrucitons?
  - It looks for officially defined pseudo-instructions, such as `ror` and `move`
  - It looks for special cases where the operand is incorrect for the operation and tries to handle it gracefully

Question

- Which lines below are pseudo-instructions (MIPS Assembly Language); that is, not TAL?
  1. addi $t0, $t1, 40000
  2. beq $s0, 10, Exit
  3. sub $t0, $t1, 1

A. 1 only
B. 2 only
C. 3 only
D. 1 and 2
E. 2 and 3
F. All of the above

Question

- Which lines below are pseudo-instructions (MIPS Assembly Language); that is, not TAL?
  1. addi $t0, $t1, 40000 **40,000 > +32,767 thus need lui, ori**
  2. beq $s0, 10, Exit **beq: both must be registers**
  3. 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. 1 only
B. 2 only
C. 3 only
D. 1 and 2
E. 2 and 3
F. All of the above

Summary

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

- Machine Language Instruction:
  - 32 bits representing a single instruction
- Branches use PC-relative addressing,
  Jumps use absolute addressing
- Disassembly is easy: starts by decoding `opcode` field
- Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)

Summary

- To understand the MIPS architecture and be sure to get best performance, it is best to study the True Assembly Language instructions.

Review and More Information

- Textbook
  - 2.5 Representing Instructions in the computer
  - 2.10 Addressing for 32-bit immediates
  - 2.12 Translating and Starting a Program
    - Just the section on the **Assembler** with respect to pseudoinstructions (pg 124, 125, 5th edition)