Logical & Shift Operations

Adapted from CS 61C & 152 © UCB 2001

Overview

- Logical Instructions
- Shifts
- Loading/Storing Bytes
- Overflow in Arithmetic

Bitwise Operations

- Up until now, we've done arithmetic (`add`, `sub`, `addi`), memory access (`lw` and `sw`), and branches and jumps
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- New Perspective: View contents of register as 32 raw bits rather than as a single 32-bit number
- Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole
- Introduce two new classes of instructions:
  - Logical Operators
  - Shift Instructions

Logical Operators

- Two basic logical operators:
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
- In general, can define them to accept >2 inputs, but in the case of MIPS assembly, both of these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware
- Truth Table: standard table listing all possible combinations of inputs and resultant output for each

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Logical Operators (Cont'd)

- **Logical Instruction Syntax:**

  1 2,3,4

  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) second operand (register) or immediate (numerical constant)

- **Instruction Names:**

  - `and, or` — Both of these expect the third argument to be a register
  - `andi, ori` — Both of these expect the third argument to be an immediate

- **MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.**

Uses for Logical Operators

- **Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.**
- **This can be used to create a mask.**
  - **Example:**

    1011 0110 1010 0100 0011 1101 1001 1010

    mask: 0000 0000 0000 0000 0000 1111 1111 1111

  - **The result of anding these:**

    0000 0000 0000 0000 0000 1101 1001 1010

    mask last 12 bits

Uses for Logical Operators (cont'd)

- The second bit string in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bit string by masking out the rest of the string (e.g. setting it to all 0s).
- Thus, the and operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
  - In particular, if the first bit string in the above example were in $t0, then the following instruction would mask it:

```
andi   $t0,$t0,0xFFF
```


Uses for Logical Operators (cont'd)

- Similarly, note that oring a bit with 1 produces a 1 at the output while oring a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
  - For example, if $t0 contains 0x12345678, then after this instruction:

```
ori    $t0, $t0, 0xFFFF
```

  - … $t0 contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s)

Shift Instructions

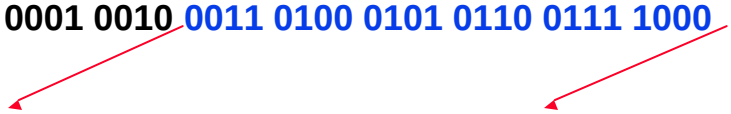
- Move (shift) all the bits in a word to the left or right by a number of bits.
  - Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

  - Example: shift left by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000



0011 0100 0101 0110 0111 1000 0000 0000

Shift Instructions (cont'd)

- Shift Instruction Syntax:

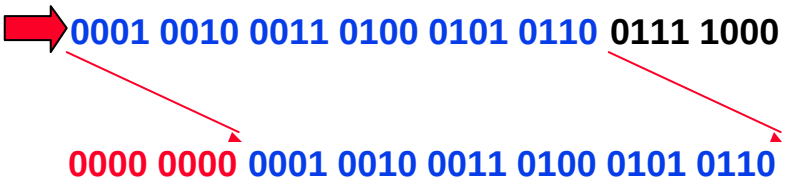
  1 2,3,4

  - where
    1) operation name
    2) register that will receive value
    3) first operand (register)
    4) shift amount (constant <= 32)

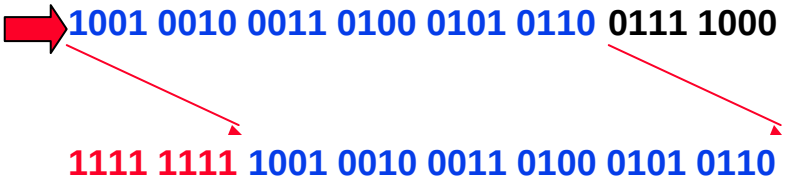
- MIPS shift instructions:
  1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
  2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
  3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (cont'd)

- Example: shift right arith by 8 bits



- Example: shift right arith by 8 bits



Uses for Shift Instructions

- Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in $t0. Simply use:

```
andi    $t0,$t0,0xFF
```

- Suppose we want to isolate byte 1 (bit 15 to bit 8) of a word in $t0. We can use:

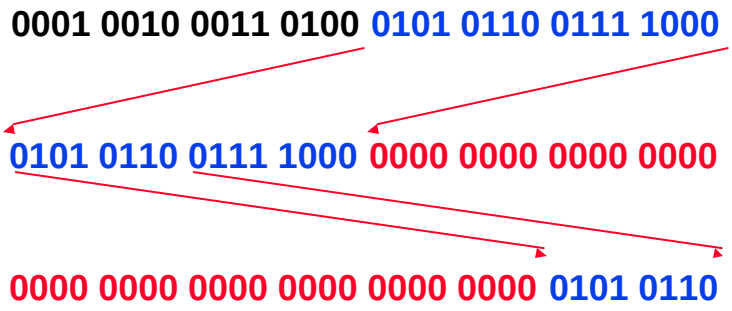
```
andi    $t0,$t0,0xFF00
```

but then we still need to shift to the right by 8 bits...

Uses for Shift Instructions

- Could use instead:

```
sll     $t0, $t0, 16
srl     $t0, $t0, 24
```



Uses for Shift Instructions

- In decimal:
  - Multiplying by 10 is same as shifting left by 1:
    - $714_{10} \times 10_{10} = 7140_{10}$
    - $56_{10} \times 10_{10} = 560_{10}$
  - Multiplying by 100 is same as shifting left by 2:
    - $714_{10} \times 100_{10} = 71400_{10}$
    - $56_{10} \times 100_{10} = 5600_{10}$
  - Multiplying by 10^n is same as shifting left by n

Uses for Shift Instructions

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - $11_2 \times 10_2 = 110_2$
    - $1010_2 \times 10_2 = 10100_2$
  - Multiplying by 4 is same as shifting left by 2:
    - $11_2 \times 100_2 = 1100_2$
    - $1010_2 \times 100_2 = 101000_2$
  - Multiplying by 2^n is same as shifting left by n

Uses for Shift Instructions

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

```
a = a*8
```

would compile to:

```
sll   $s0,$s0,3
```

(in MIPS)

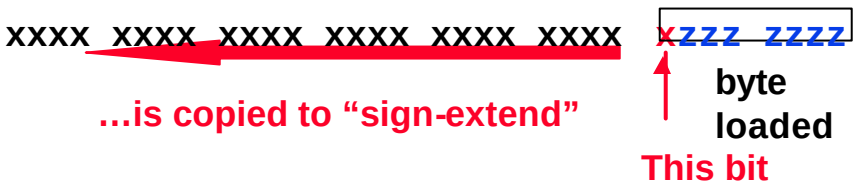
- Likewise, shift right to divide by powers of 2
  - remember to use `sra`

Loading, Storing bytes

- In addition to word data transfers (`lw`, `sw`), MIPS has byte data transfers:
  - load byte `lb`
  - store byte `sb`
  - same format as `lw`, `sw`

Loading, Storing bytes (cont'd)

- What do the other 24 bits in the 32 bit register?
  - `lb`: sign extends to fill upper 24 bits



- Normally with characters don't want to sign extend
- MIPS instruction that doesn't sign extend when loading bytes:

  load byte unsigned: `lbu`

Question

Suppose:

```
lb $s0, 100($zero) #byte@100= 0x0F
lb $s1, 200($zero) #byte@200= 0xFF
```

What are the values of $s0 and $s1?

		$s0	$s1
A:	A:	15	255
B:	B:	15	-1
C:	C:	15	-255
D:	D:	-15	255
E:	E:	-15	-1
F:	F:	-15	-255

Answer

Suppose:

```
lb $s0, 100($zero) #byte@100= 0x0F
lb $s1, 200($zero) #byte@200= 0xFF
```

What are the values of $s0 and $s1?

		$s0	$s1
☺ A:	A:	15	255
B:	B:	15	-1
C:	C:	15	-255
D:	D:	-15	255
E:	E:	-15	-1
F:	F:	-15	-255

```
?.. ???? ???? ????   s0
?.. ???? 0000 1111   15
0.. 0000 0000 1111

?.. ???? ???? ????   s1
?.. ???? 1111 1111   -1
1.. 1111 1111 1111
```

Things to Remember

- **Logical and Shift Instructions**
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use shift left logical, `sll`, for multiplication by powers of 2
  - Use shift right arithmetic, `sra`, for division by powers of 2.
- **New Instructions:**
  `and, andi, or, ori, sll, srl, sra`

Things to Remember

- **MIPS Signed v. Unsigned is an "overloaded" term**
  - Do/Don't sign extend (lb, lbu)
  - Don't overflow (addu, addiu, subu, multu, divu)
  - Do signed/unsigned compare (slt,slti/sltu,sltiu)

MIPS Instruction Representation

Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers.
  2) Therefore, entire programs can be stored in memory to be read or written just like numbers (data).
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for Macintosh and IBM PC
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today

Instructions as Numbers

- 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?
  - Remember: Computer only understands 1s and 0s, so "`add $t0,$0,$0`" is meaningless.
  - MIPS wants simplicity: since data is in words, make instructions be words too

Instructions as Numbers (cont'd)

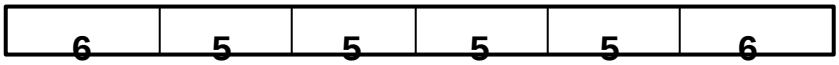
- One word is 32 bits, so divide instruction word into "fields"
- Each field tells computer something about instruction
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - R-format
  - I-format
  - J-format

Instruction Formats

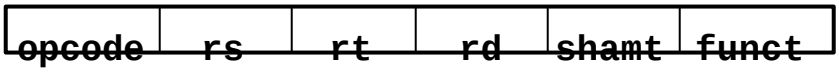
- I-format: used for instructions with immediates, `lw` and `sw` (since the offset counts as an immediate), and the branches (`beq` and `bne`),
  - (but not the shift instructions; later)
- J-format: used for `j` and `jal` (see Friday)
- R-format: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way.

R-Format Instructions

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



- For simplicity, each field has a name:



- Important: On these slides and in the 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

- What do these field integer values tell us?
  - `opcode`: partly specifies what instruction it is
    - Note: 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?
    - Answer: We'll answer this later.

R-Format Instructions (cont'd)

- 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 (cont'd)

- **Notes about register fields:**
  - 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 that we'll see later, e.g.,
    - `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 (cont'd)

- **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 back cover of textbook.**

R-Format Example (1/2)

- MIPS Instruction:

```
add    $8,$9,$10
```

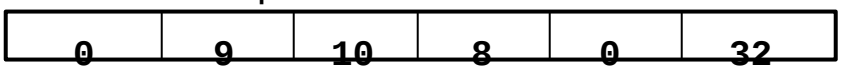
opcode	0 (look up in table)
funct	32 (look up in table)
rs	9 (first *operand*)
rt	10 (second *operand*)
rd	8 (destination)
shamt	0 (not a shift)

R-Format Example (2/2)

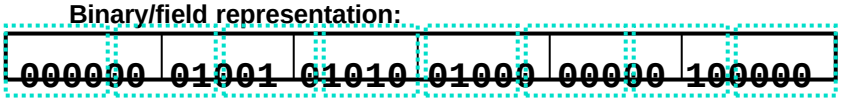
- MIPS Instruction:

```
add    $8,$9,$10
```

Decimal/field representation:



Binary/field representation:



hex

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

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

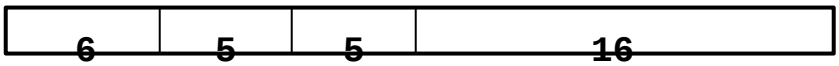
- Called a Machine Language Instruction

I-Format Instructions (1/5)

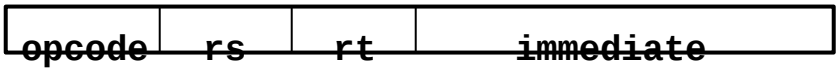
- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:
  - First notice that, if instruction has immediate, then it uses at most 2 registers.

I-Format Instructions (2/5)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 16 = 32 bits



- Again, each field has a name:



- Key Concept: Only one field is inconsistent with R-format. Most importantly, opcode is still in same location.

I-Format Instructions (3/5)

- What do these fields mean?
  - **opcode**
    Same as before except that, since there is no `funct` field, `opcode` uniquely specifies an instruction in I-format
  - This also answers the previous question

    Q: Why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field?

    A: In order to be consistent with other formats.

I-Format Instructions (4/5)

- More fields:
  - `rs`: specifies the *only* 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")

I-Format Instructions (5/5)

- 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.

I-Format Example (1/2)

- MIPS Instruction:

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

`opcode`	8 (look up in table)
`rs`	22 (register containing operand)
`rt`	21 (target register)
`immediate`	-50 (by default, this is decimal)

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$_{ten}$

I-Format Problems (1/2)

- Problem:
  - Chances are that `addi`, `lw`, `sw` and `slti` will use immediates small enough to fit in the immediate field
  - What if too big?
    - We need a way to deal with a 32-bit immediate in *any* I-format instruction
- Solution:
  - Handle it in software + new instruction
  - Don't change the current instructions; instead, add a …
- … 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

I-Format Problems (2/2)

- So how does `lui` help us?
  - Example:

```
addi   $t0,$t0, 0xABABCDCD
```

    becomes:

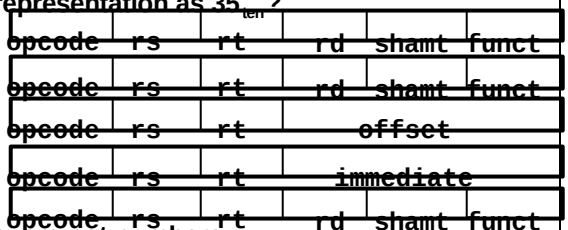
```
lui    $at, 0xABAB
ori    $at, $at, 0xCDCD
add    $t0, $t0, $at
```

  - Now each I-format instruction has only a 16-bit immediate.
  - Wouldn't it be nice if the assembler would this for us automatically?
    - Pseudoinstructions …

Question

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
F. Hmmm … Instructions are not numbers



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	

Answer

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

F. Hmmm … Instructions are not numbers

0	0	0	0	0	32
0	16	16	16	0	35
35	0	0			0
8	0	0			35
0	0	0	0	0	35

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	

In conclusion, …

- 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`).
- 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 32-bit numbers.

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 …

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

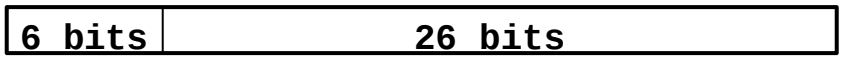
- Does the value in branch field change if we move the code?
- What do we do if its > 2^15 instructions?
- Since its limited to +- 2^15 instructions, doesn't this generate lots of extra MIPS instructions?
- Why do we need all these addressing modes? Why not just one?

J-Format Instructions (1/5)

- For branches, we assumed that we won't want to branch too far, so we can specify only the *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:



- 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.
- So, we can specify 28 bits of the 32-bit address.

J-Format Instructions (4/5)

- **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: II 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

- 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 (cont'd)

- The six machine language instructions in binary:

```
00000000000000000001000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000100010000010000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

- Next step: identify opcode and format

Decoding Example (cont'd)

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

Format:

```
R 00000000000000000001000000100101
R 00000000000001010100000000101010
I 00010001000000000000000000000011
R 00000000010001000001000000100000
I 00100001010010111111111111111111
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 (cont'd)

- 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 (cont'd)

- 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 (cont'd)

- 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 (cont'd)

- 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)

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.

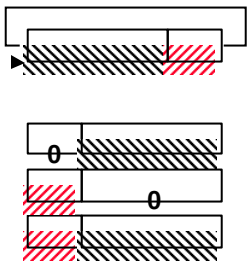
Example Pseudoinstructions

- **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.**

Question

- 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

Answer

- 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

40,000 > +32,767 => `lui,ori`

beq: both must be registers

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

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 Þ data transfer in asm.

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

*p = ...; Þ 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`

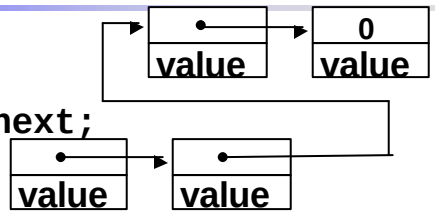
```
p = &c;  /* p gets 0x10000000 */
x = *p;  /* x gets 100 */
*p = 200; /* c gets 200 */
```

```
# 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
```