Overview - Instruction Representation

- Big idea: stored program
  - consequences of stored program
- Instructions as numbers
- Instruction encoding
- MIPS instruction format for Add instructions
- MIPS instruction format for Immediate, Data transfer instructions

Big Idea: 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 Macintoshes and PCs
- 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

- 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`
- **R-format**: used for all other instructions

R-Format Instructions

- 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**: 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**: 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?
    - Answer: We’ll answer this later.

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

R-Format Instructions

• 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

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

R-Format Instructions

• 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.,
    - **mul** 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

- **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.

R-Format Example

- **MIPS Instruction:**
  - `add $8, $9, $10`
  - `opcode = 0` (look up in table in book)
  - `funct = 32` (look up in table in book)
  - `rd = 8` (destination)
  - `rs = 9` (first operand)
  - `rt = 10` (second operand)
  - `shamt = 0` (not a shift)

R-Format Example

- **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 10000`
  - Hex representation: 012A 4020
  - Decimal representation: 19,546,144
  - Called a *Machine Language Instruction*
I-Format Instructions

• 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.

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.

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 with other formats.
  • 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

• The Immediate Field:
  • `addi`, `slt`, `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 `slt` instruction.
  • We'll see what to do when the number is too big in our next lecture.

I-Format Example

• **MIPS Instruction:**
  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)

I-Format Example (2/2)

• **MIPS Instruction:**
  addi $21,$22,-50

  Decimal/field representation:
  | 8 | 22 | 21 | -50 |

  Binary/field representation:
  001000 10110 10101 111111111001110

  Hexadecimal representation: 22D5 FFCF
  Decimal representation: 84,449,998
I-Format Problems

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
- 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 $2^5 \leq n < 2^6$ (i.e., with a 1 in the 5th bit and 0s in the upper 2 bytes) as it does for numbers that are too large. ⇒

Lec 2.23

I-Format Problems

Problem 1:
- 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?
- We need a way to deal with a 32-bit immediate in any I-format instruction.

Lec 2.23

I-Format Problems

Solution to Problem 1:
- 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 specified register
  - sets lower half to 0s

Lec 2.24
I-Format Problems

- Solution to Problem 1 (continued):
  - 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 do this for us automatically? (later)

Branches: PC-Relative Addressing

- Use I-Format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

  - opcode specifies beq, 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.

- 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: may want to branch to anywhere in memory, but a branch often changes PC by a small amount
Branches: PC-Relative Addressing

- 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

- 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

- Branch Calculation:
  - If we don’t take the branch:
    \[ PC = PC + 4 \]
    \[ PC+4 = \text{byte address of next instruction} \]
  - If we do take the branch:
    \[ PC = (PC + 4) + (\text{immediate} \times 4) \]
- 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

MIPS Code:
Loop: beq $9, $0, End
addi $8, $8, $10
addi $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 = ???

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

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

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

Lec 2.31

Lec 2.32

Lec 2.33
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?
- Since it's limited to ± $2^{15}$ instructions, doesn't this generate lots of extra MIPS instructions?

J-Format Instructions

- 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

- 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

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

Now specify 28 bits of a 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
  - only if straddle a 256 MB boundary
- 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

Summary:
- New PC = ( PC[31..28], target address, 00 )
- Understand where each part came from!
- Note: ( , , ) means concatenation
  ( 4 bits , 26 bits , 2 bits ) = 32 bit address
  - ( 1010, 11111111111111111111111111, 00 ) = 10101111111111111111111111111100
  - Note: Book uses ||
In semi-conclusion...

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

<table>
<thead>
<tr>
<th>R</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>opcode</td>
<td>target address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS Addressing Modes**

1. Immediate addressing
2. Register addressing
3. Base addressing
4. PC-relative addressing
5. Pseudodirect addressing

**Add / Subtract**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-type</strong></td>
<td>add</td>
<td>add $1, $2, $3</td>
<td>$1 = $2 + $3</td>
</tr>
<tr>
<td>sub</td>
<td>sub $1, $2, $3</td>
<td>$1 = $2 - $3</td>
<td>3 operands, overflow detected</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1, $2, $3</td>
<td>$1 = $2 + $3</td>
<td>3 operands, no exception</td>
</tr>
<tr>
<td>sub unsigned</td>
<td>subu $1, $2, $3</td>
<td>$1 = $2 - $3</td>
<td>3 operands, no exception</td>
</tr>
<tr>
<td><strong>I-type</strong></td>
<td>add immediate</td>
<td>add $1, $2, 10</td>
<td>$1 = $2 + 10</td>
</tr>
<tr>
<td>add immediate unsigned</td>
<td>addu $1, $2, 10</td>
<td>$1 = $2 + 10</td>
<td>constant, no exception, sign extension</td>
</tr>
</tbody>
</table>
And, Or, Xor, Nor

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>and</td>
<td>$1 &amp; {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>3 operands; Logical AND</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>$1 \lor {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>3 operands; Logical OR</td>
</tr>
<tr>
<td></td>
<td>xor</td>
<td>$1 \land {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>3 operands; Logical XOR</td>
</tr>
<tr>
<td></td>
<td>nor</td>
<td>$1 \lor {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>3 operands; Logical NOR</td>
</tr>
<tr>
<td>I-type</td>
<td>and</td>
<td>$1 &amp; {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>$1 \lor {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>Logical OR reg, constant</td>
</tr>
<tr>
<td></td>
<td>xor</td>
<td>$1 \land {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>Logical XOR reg, constant</td>
</tr>
</tbody>
</table>

- No sign extension for logical immediate instructions

Shift Instructions

<table>
<thead>
<tr>
<th>R-type</th>
<th>shift left logical</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>left</td>
<td>$1 \ll {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td></td>
<td>left</td>
<td>$1 \ll {\text{reg, variable} } = {\text{reg, variable} }$</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td></td>
<td>right</td>
<td>$1 \gg {\text{reg, constant} } = {\text{reg, constant} }$</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td></td>
<td>right</td>
<td>$1 \gg {\text{reg, variable} } = {\text{reg, variable} }$</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td></td>
<td>right</td>
<td>$1 \gg {\text{reg, variable} } = {\text{reg, variable} }$</td>
<td>Shift right: Sign extension</td>
</tr>
<tr>
<td></td>
<td>right</td>
<td>$1 \gg {\text{reg, variable} } = {\text{reg, variable} }$</td>
<td>Shift right: Sign extension</td>
</tr>
</tbody>
</table>

Data Transfer Instructions

<table>
<thead>
<tr>
<th>I-type</th>
<th>store word</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sh $1, 21($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>sb $1, 21($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>lh $1, 22($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Half word from register to memory</td>
</tr>
<tr>
<td></td>
<td>lw $1, 20($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>lh $1, 22($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Half word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lw $1, 20($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lh $1, 22($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Half word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lw $1, 20($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lh $1, 22($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Half word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lw $1, 20($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lh $1, 22($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Half word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lw $1, 20($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lh $1, 22($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Half word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lw $1, 20($2)</td>
<td>$1 \rightarrow {\text{reg, memory} }$</td>
<td>Word from memory to register</td>
</tr>
</tbody>
</table>
Multiplying and Dividing

- **R-type instructions**
  - **Multiply / divide**
    - `mul rt, rs`
    - `multu rt, rs`
    - `div rt, rs`
    - `divu rt, rs`
  - **Move from HI or LO**
    - `mfhi rd`
    - `mflo rd`
  - **Move to HI or LO**
    - `mtfi rd`
    - `mtfl rd`

**Registers**

- HI
- LO

---

MIPS Conditional Branch

- **Compare and Branch**
  - `BEQ rs, rt, offset` if `R[rs] == R[rt]` then PC-relative branch
  - `BNE rs, rt, offset` if `R[rs] != R[rt]` then PC-relative branch

- **Compare to zero and Branch**
  - `BLEZ rs, offset` if `R[rs] <= 0` then PC-relative branch
  - `BGTZ rs, offset` if `R[rs] > 0` then PC-relative branch
  - `BLTZ` branch and link

- **I-type and PC-relative addressing**
  - Immediate portion specifies the number of instructions

---

MIPS Compare and Jump Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>slt $1, $2, $3</code></td>
<td>if $(2) &lt; $(3) then $(1) = 1`</td>
<td>Set less than</td>
</tr>
<tr>
<td><code>slti $1, $2, 10</code></td>
<td>if $(2) &lt; 10 then $(1) = 1`</td>
<td>Immediate signed less than</td>
</tr>
</tbody>
</table>

- **J-type uses Pseudodirect addressing**

---

J-type uses Pseudodirect addressing.