CS 61C: Great Ideas in Computer Architecture (Machine Structures)

MIPS Instruction Formats

Instructors:
Michael Greenbaum

http://inst.eecs.Berkeley.edu/~cs61c/su11
Agenda

• Review: Register Conventions
• MIPS Logical Instructions
• Administrivia
• Stored Program Concept
• R-Format
• Break
• I-Format, J-Format
• Conclusion
Parents leaving for weekend analogy (1/5)

- Parents *(main)* leaving for weekend
- They *(caller)* give keys to the kid *(callee)* with the rules *(calling conventions)*:
  - You can trash the temporary room(s), like the den and basement *(registers)* if you want, we don’t care about it
  - BUT you’d better leave the rooms *(registers)* that we want to save for the guests untouched. “these rooms better look the same when we return!”
- Who hasn’t heard this in their life?
Parents leaving for weekend analogy (2/5)

• Kid now “owns” rooms (registers)
• Kid wants to use the saved rooms for a wild, wild party (computation)
• What does kid (callee) do?
  – Kid takes what was in these rooms and puts them in the garage (memory)
  – Kid throws the party, trashes everything (except garage, who ever goes in there?)
  – Kid restores the rooms the parents wanted saved after the party by replacing the items from the garage (memory) back into those saved rooms
Parents leaving for weekend analogy (3/5)

• Same scenario, except before parents return and kid replaces saved rooms...

• Kid’s friend (another callee) wants the house for a party when the kid is away!
  – Kid (now the caller) has left valuable stuff (data) all over.
  – Kid knows that friend might trash the place destroying valuable stuff!
  – Kid remembers rule parents taught and now becomes the “heavy” (caller), instructing friend (callee) on good rules (conventions) of house.
Parents leaving for weekend analogy (4/5)

• If kid had data in temporary rooms (which were going to be trashed), there are three options:
  – Move items directly to garage (memory)
  – Move items to saved rooms whose old contents have already been moved to the garage (memory)
  – Optimize lifestyle (code) so that the amount you’ve got to schlep stuff back and forth from garage (memory) is minimized.
    • Mantra: “Minimize register footprint”

• Otherwise: “Dude, where’s my data?!?”
Parents leaving for weekend analogy (5/5)

- **Friend** now “owns” rooms *(registers)*
- Friend wants to use the saved rooms for a wild, wild party *(computation)*
- What does friend *(callee)* do?
  - Friend takes what was in these rooms and puts them in the garage *(memory)*
  - Friend throws the party, trashes everything (except garage)
  - Friend restores the rooms the kid wanted saved after the party by replacing the items from the garage *(memory)* back into those saved rooms
Example: Using Saved Registers

myFunc: # Decides it will use two saved registers
  addiu $sp,$sp,-12
  sw $ra,8($sp)
  sw $s0,4($sp) # saves the old values
  sw $s1,0($sp) # of the saved registers
  ... # do stuff with $s0 and $s1
  jal func1
  ... # $s0 and $s1 unchanged by function
  jal func2 # calls, can keep using them
  ...
  lw $ra,8($sp)
  lw $s0,4($sp) # but must do your part in following
  lw $s1,0($sp) # convention and restore $s registers
  addiu $sp,$sp,12 # to their original values
  jr $ra
Agenda

- Review: Register Conventions
- MIPS Logical Instructions
- Administrivia
- MIPS Instruction Formats
- R-Format
- Break
- I-Format, J-Format
- Conclusion
MIPS Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

<table>
<thead>
<tr>
<th>Logical operations</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
</tbody>
</table>
Implementing bitwise ‘not’

• NOR (not OR) truth table:

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• NORing an input with itself inverts that input.

• Bitwise ‘not’ in MIPS:
  - `nor $s0,$s1,$s1`
  - `nor $s0,$s1,$0` also works. Why?
Shifting

• Shift left logical moves n bits to the left (insert 0s into empty bits)
  – Same as multiplying by $2^n$ for two’s complement number
• For example, if register $s0$ contained
  0000 0000 0000 0000 0000 0000 0000 1001\text{two} = 9\text{ten}
• If executed sll $s0$, $s0$, 4, result is:
  0000 0000 0000 0000 0000 0000 0000 1001 0000\text{two} = 144\text{ten}
• And $9\text{ten} \times 2\text{ten}^4 = 144\text{ten}$
• Shift right logical moves n bits to the right (insert 0s into empty bits)
  – NOT same as dividing by $2^n$ (negative numbers fail)
Shifting

- *Shift right arithmetic* \((\text{sra})\) moves \(n\) bits to the right (extends highest order sign bit into empty bits)
- For example, if register \(\$s0\) contained
  \[
  0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 1001_{\text{two}} = 25_{\text{ten}}
  \]
- If executed \texttt{sra} \(\$s0, \$s0, 4\), result is:
  \[
  0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
  \]
Shift right arithmetic ($sra$) moves n bits to the right (extends highest order sign bit into empty bits)

- For example, if register $s0$ contained $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{\text{two}} = -25_{\text{ten}}$
- If executed $sra\ s0,\ s0,\ 4$, result is: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2_{\text{ten}}$
- Unfortunately, this is still NOT same as dividing by $2^n$
  - Fails for odd negative numbers – rounds the wrong way
  - -1 shifted right arithmetically by 1 is STILL -1.
  - C arithmetic semantics is that division should round towards 0
Agenda

• Review: Register Conventions
• MIPS Logical Instructions
• **Administrivia**
• Stored Program Concept
• R-Format
• **Break**
• I-Format, J-Format
• Conclusion
Administrivia

• HW2 due Sunday, 11:59:59.
• Project 1 posted by tomorrow, due 7/10.
  – No homework that week.
• OH Locations updated on course webpage
  – Mine: Held in 651 Soda (6th floor alcove)
  – Justin, Alvin: Held in 411 Soda (4th floor alcove)
• I’ve put in room requests for the Midterm and Final, any start time available ranging from 9am to 12pm. We’ll see what we get...
Agenda

• Review: Register Conventions
• MIPS Logical Instructions
• Administrivia
• Stored Program Concept
• R-Format
• Break
• I-Format, J-Format
• Conclusion
Big Idea: Stored-Program Concept

• Encode your instructions as binary data.
  – Therefore, entire programs can be stored in memory to be read or written just like 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, 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
Consequence #2: Everything Addressed

• Programs are distributed in binary form
  – Programs bound to specific instruction set
  – Different version for (old) Macintoshes and PCs

• New machines want to run old programs ("binaries") as well as programs compiled to new instructions

• Leads to “backward compatible” 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 (1/2)

• 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 (2/2)

• One word is 32 bits, so divide instruction word into “fields”.
• Each field tells processor 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
Levels of Representation/Interpretation

High Level Language Program (e.g., C) → Compiler → Assembly Language Program (e.g., MIPS) → Assembler → Machine Language Program (MIPS) → Machine Interpretation → Hardware Architecture Description (e.g., block diagrams) → Architecture Implementation → Logic Circuit Description (Circuit Schematic Diagrams)

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

lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)

Anything can be represented as a number, i.e., data or instructions
Agenda

• Review: Register Conventions
• MIPS Logical Instructions
• Administrivia
• Stored Program Concept
• R-Format
• Break
• I-Format, J-Format
• Conclusion
R-Format Instructions (1/5)

• Define “fields” of the following number of bits each: $6 + 5 + 5 + 5 + 5 + 6 = 32$

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

• For simplicity, each field has a name:

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

• Important: Each field is viewed as its own 5 or 6 bit unsigned 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:

• With some exceptions:
  – **rs** (Source Register): used to specify register containing first operand
  – **rt** (Target Register): used to specify register containing second operand (note that name is misleading)
  – **rd** (Destination Register): used to specify register which will receive result of computation

• **eg.** `add rd, rs, rt`
R-Format Instructions (4/5)

• 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.
  – 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 (see COD)
# Registers by Number (left column)

<table>
<thead>
<tr>
<th>Category</th>
<th>Register Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The constant 0</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>Reserved for Assembler</td>
<td>$1$</td>
<td>$at$</td>
</tr>
<tr>
<td>Return Values</td>
<td>$2$-$3$</td>
<td>$v0$-$v1$</td>
</tr>
<tr>
<td>Arguments</td>
<td>$4$-$7$</td>
<td>$a0$-$a3$</td>
</tr>
<tr>
<td>Temporary</td>
<td>$8$-$15$</td>
<td>$t0$-$t7$</td>
</tr>
<tr>
<td>Saved</td>
<td>$16$-$23$</td>
<td>$s0$-$s7$</td>
</tr>
<tr>
<td>More Temporary</td>
<td>$24$-$25$</td>
<td>$t8$-$t9$</td>
</tr>
<tr>
<td>Used by Kernel</td>
<td>$26$-$27$</td>
<td>$k0$-$k1$</td>
</tr>
<tr>
<td>Global Pointer</td>
<td>$28$</td>
<td>$gp$</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>$29$</td>
<td>$sp$</td>
</tr>
<tr>
<td>Frame Pointer</td>
<td>$30$</td>
<td>$fp$</td>
</tr>
<tr>
<td>Return Address</td>
<td>$31$</td>
<td>$ra$</td>
</tr>
</tbody>
</table>

(From MIPS green sheet)
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 green insert in COD (You will be given a copy for all exams)
R-Format Example (1/2)

- Pseudocode (OPERATION on green sheet)
  \[
  \text{add} \quad R[rd] = R[rs] + R[rt]
  \]

- MIPS Instruction:
  \[
  \text{add} \quad \$8,\$9,\$10
  \]

  \textbf{opcode} = 0 (look up on green sheet)
  \textbf{funct} = 32 (look up on green sheet)
  \textbf{rd} = 8 (destination)
  \textbf{rs} = 9 (first \textit{operand})
  \textbf{rt} = 10 (second \textit{operand})
  \textbf{shamt} = 0 (not a shift)
R-Format Example (2/2)

- MIPS Instruction:
  \texttt{add \$8,\$9,\$10}

Decimal number per field representation:

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
0 & 9 & 10 & 8 & 0 & 32 \\
\end{array}
\]

Binary number per field representation:

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

hex representation: \(012A\ 4020\)\text{_{hex}}

decimal representation: \(19,546,144\)\text{_{ten}}

Called a \textbf{Machine Language Instruction}

(This is part of the process of assembly)
Everything in a Computer is Just a Binary Number

• Up to program to decide what data means
• Example 32-bit data shown as binary number:
  0000 0000 0000 0000 0000 0000 0000 0000 \_two

What does it mean if its treated as

1. Signed integer
2. Unsigned integer
3. ASCII characters
4. Unicode characters
5. MIPS instruction
Agenda

• Review: Register Conventions
• MIPS Logical Instructions
• **Administrivia**
• Stored Program Concept
• R-Format
• **Break**
• I-Format, J-Format
• Conclusion
Agenda

• Review: Register Conventions
• MIPS Logical Instructions
• **Administrivia**
• Stored Program Concept
• R-Format
• **Break**
• I-Format, J-Format
• Conclusion
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

- Again, each field has a name:

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

- **Key Concept**: Three fields are consistent with R-Format instructions. 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 today...
I-Format Example (1/2)

• Pseudocode (OPERATION on green sheet)
  \[
  \text{addi} \quad R[rt] = R[rs] + \text{SignExtImm}
  \]

• MIPS Instruction:
  \[
  \text{addi} \quad $21,$22,-50
  \]

  \text{opcode} = 8 \text{ (look up on green sheet)}
  \text{rs} = 22 \text{ (register containing operand)}
  \text{rt} = 21 \text{ (target register)}
  \text{immediate} = -50 \text{ (by default, this is decimal)}
I-Format Example (2/2)

- MIPS Instruction:
  ```
  addi $21,$22,-50
  ```

<table>
<thead>
<tr>
<th>Decimal/field representation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary/field representation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
</tr>
</tbody>
</table>

hexadecimal representation: 22D5 FFCE

decimal representation: 584,449,998
Peer Instruction

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Instruction</th>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>Shamt</th>
<th>Funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>b)</td>
<td>add $0, $0, $0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>g)</td>
<td>subu $s0, $s0, $s0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>p)</td>
<td>lw $0, 0($0)</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y)</td>
<td>addi $0, $0, 35</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r)</td>
<td>subu $0, $0, $0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>35</td>
</tr>
</tbody>
</table>

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
Which instruction has the same representation as \(35_{ten}\)?

- b) add $0, $0, $0
- g) subu $s0, $s0, $s0
- p) lw $0, 0($0)
- y) addi $0, $0, 35
- r) subu $0, $0, $0

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 (1/4)

• 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. Remember, the u means don’t signal overflow, not signed vs unsigned integers!
  – **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 0s in the upper 2 bytes) as it does for numbers that are too large. $\Rightarrow$
I-Format Problem (2/4)

• Problem:
  – Chances are that \texttt{addi, lw, sw} and \texttt{slti} will use immediates small enough to fit in the immediate field.
  – ...but what if it’s too big?
  – And what about bitwise logic operations on a full 32 bit immediate?
  – We need a way to deal with a 32-bit immediate in any I-format instruction.
I-Format Problem (3/4)

• 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:
  
  \[
  \text{lui register, immediate}
  \]
  
  – stands for \text{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 0s
I-Format Problems (4/4)

• Solution to Problem (continued):
  – So how does `lui` help us?
  – Example:
    
    ```
    addi $t0,$t0, 0xABABCDCD
    ...
    ```

    becomes

    ```
    lui $at, 0xABAB
    ori $at, $at, 0xCD
    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? (later)
Branches: PC-Relative Addressing (1/5)

- Use I-Format

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

- `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.
Branch Calculation:

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

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 (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 = ???
Branch Example (2/3)

• MIPS Code:

  Loop:  beq  $9,$0,End
         addu  $8,$8,$10
         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:**
  
  ```mips
  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 0000000000000000011
Questions on PC-addressing

• Does the value in branch immediate 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 forming a memory address)? 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 two “fields” of these bit widths:
  - 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.
  - Collapse 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 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)

• 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 (5/5)

• Summary:
  – New PC = \{ (PC+4)[31..28], target address, 00 \}

• Understand where each part came from!

• Note: \{ , , \} means concatenation
  \{ 4 bits , 26 bits , 2 bits \} = 32 bit address
  – \{ 1010, 11111111111111111111111111111111, 00 \} = 101011111111111111111111111111100
  – Note: Book uses \|\|
When combining two C files into one executable, recall we can compile them independently & then merge them together. When merging two or more binaries:

1) **Jump** insts don’t require any changes.
2) **Branch** insts don’t require any changes.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>FF</td>
</tr>
<tr>
<td>b)</td>
<td>FT</td>
</tr>
<tr>
<td>c)</td>
<td>TF</td>
</tr>
<tr>
<td>d)</td>
<td>TT</td>
</tr>
<tr>
<td>e)</td>
<td>dunno</td>
</tr>
</tbody>
</table>
When combining two C files into one executable, recall we can compile them independently & then merge them together. When merging two or more binaries:

1. **Jump** insts don’t require any changes.
2. **Branch** insts don’t require any changes.

Peer Instruction Question Answer

- a) FF
- b) FT
- c) TF
- d) TT
- e) dunno

Answer: b) FT
“And in Conclusion, …”

- **MIPS Machine Language Instruction:** Encode an instruction in 32 bits!

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

- Branches use PC-relative addressing, Jumps use absolute addressing.
- Disassembly is simple and starts by decoding **opcode** field. (more in a week)