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

*More MIPS Instruction Formats, Assembly, Linking*

Instructors:
Michael Greenbaum

---

**Agenda**
- I and J Instruction Formats
- Administrivia
- The Assembler
- Break
- Linking
- Conclusion

---

**Levels of Representation/Interpretation**

<table>
<thead>
<tr>
<th>High Level Language Program (e.g., C)</th>
<th>lw $t0, 0($2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Language Program (e.g., MIPS)</td>
<td>lw $t1, 4($2)</td>
</tr>
<tr>
<td>Machine Language Program (MIPS)</td>
<td>sw $t1, 0($2)</td>
</tr>
<tr>
<td>Machine Interpretation</td>
<td>sw $t0, 4($2)</td>
</tr>
<tr>
<td>Hardware Architecture Description</td>
<td>i.e., data or instructions</td>
</tr>
<tr>
<td>Logic Circuit Description</td>
<td>Circuit Schematic Diagrams</td>
</tr>
</tbody>
</table>

**Review**

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

<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>
</tbody>
</table>

- Reading an instruction starts by decoding the opcode field.
- J-Format – Coming up soon...

---

**Data Transfer Instructions**

- The data transfer instructions (lw, sw, lb, sb, etc.) specify two registers and a constant (the offset).
- Represented with I-Format
- Example:
  ```
  lw \$t0 -> 4($t1)
  ```
  - opcode = 35 => 100011
  - rs = 9 (\$t1) => 01001
  - rt = 8 (\$0) => 01000
  - imm = -4 => 1111111111111110

| 100011 | 01001 | 01000 | 1111111111111110 |

---

**16-Bit Immediate?**

- 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 well get wrong answers?
    - Nope, it means assembler has to handle any unsigned immediate $2^n \leq n < 2^{16}$ (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. ☐
16-Bit Immediate?

- Problem:
  - 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? What about logic operations?
  - Want to support up to a 32-bit immediate
    - Allows full bitwise logic operations
    - Not much point in supporting larger than processor word size

- 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:
  - `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 register
    - sets lower half to 0s

16-Bit Immediate?

- Solution to Problem (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 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.

Branches: PC-Relative Addressing (5/5)

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

  Loop:
  
  beq $9, $0, End
  addu $8, $8, $10
  addiu $9, $9, -1
  j Loop

  End:

  decimal representation:
  
  \[ 4 \quad 9 \quad 0 \quad 3 \]

  binary representation:
  
  \[ 000100 \quad 01001 \quad 00000 \quad 0000000000000011 \]

Questions on PC-Relative 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?
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)

31
6 bits
26 bits

• Define two “fields” of these bit widths:

• As usual, each field has a name:

<table>
<thead>
<tr>
<th>opcode</th>
<th>target address</th>
</tr>
</thead>
</table>

• 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, 111111111111111111111111111111111111111111111111111111111111100 }
  – Note: Book uses ||

Peer Instruction Question

When combining two C files into one executable, recall we can compile and assemble 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.
When combining two C files into one executable, recall we can compile and assemble 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

When calling function in another file:
Also, absolute addresses of labels may change as multiple files are merged together.

Branch addresses are relative.

Agenda

- I and J Instruction Formats
- Administrivia
- The Assembler
- Break
- Linking
- Conclusion

Administrivia

- Project 1 has been posted
  - Due Sunday at midnight
  - Part 2 posted today – A short but “interesting” MIPS program to run on your simulator.
  - How are things going?
- The Midterm is next Friday.
  - Exact location and time still TBD.
  - Review session held next Monday.

Agenda

- I and J Instruction Formats
- Administrivia
- The Assembler
- Break
- Linking
- Conclusion

Assembler

- Input: Assembly Language File (e.g., foo.s for MIPS)
- Outputs: Object File (e.g., foo.o for MIPS)
  - Produces Machine Language
  - Replace Pseudoinstructions
  - Reads and Uses Directives
  - Produces various information tables...
Pseudoinstructions (1/3)

• MIPS “Instructions” that are convenient for an assembly programmer to use
  – Programmer meaning either a human assembly writer or a compiler.
• Get translated by the assembler into real instructions.
• Some examples follow...

Example Pseudoinstructions

• Register Move
  move reg2, reg1
  Translates 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

Example Pseudoinstructions

• Load Address: How do we get the address of an instruction or global variable into a register?

  la reg, label
  Again if value fits in 16 bits:
  addi reg, $zero, label_value
  else:
  lui reg, upper_16_bits_of_value
  ori reg, $zero, lower_16_bits

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
  sll $0, $0, 0

Example Pseudoinstructions

• Wrong operation for operand
  addu reg, reg, value # should be addiu
  If value fits in 16 bits, addu is changed to:
  addiu reg, reg, value
  else:
  lui $at, upper_16_bits_of_value
  ori $at, $at, lower_16_bits
  addu reg, reg, $at

• Any 32-bit immediates will get translated into an appropriate lui/ori combination!
Pseudoinstructions (3/3)

• **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 (which exist in the MIPS ISA) that can actually get directly translated into a single machine language instruction (32-bit binary string). Green sheet is **TAL**
• A program must be converted from MAL into TAL before translation into 1s & 0s.

Assembler Directives (p. B-5 to B-7)

• Give directions to assembler, but do not produce machine instructions
  • `.text`: Subsequent items put in user text segment
  • `.data`: Subsequent items put in user data segment
  • `.globl sym`: declares `sym` global and can be referenced from other files
  • `.asciiz str`: Store the string `str` in memory and null-terminate it
  • `.word wi...wn`: Store the `n` 32-bit quantities in successive memory words

Producing Machine Language (1/3)

• **Simple Case**
  — Arithmetic, Logical, Shifts, and so on.
  — All necessary info is within the instruction already.
• **What about Branches?**
  — PC-Relative
  — So once pseudoinstructions are replaced by real ones, we know by how many instructions to branch.
• So these can be handled.
  — Extra for experts: What about long branches?

Producing Machine Language (2/3)

• **“Forward Reference” problem**
  — Branch instructions can refer to labels that are “forward” in the program:
    ```
    or $v0, $0, $0
    L1:  slt $t0, $0, $a1
    beq $t0, $0, L2
    addi $a1, $a1, -1
    j   L1
    L2:  add $t1, $a0, $a1
    ```
  — Solved by taking 2 passes over the program.
    • First pass remembers position of labels
    • Second pass uses label positions to generate code

Producing Machine Language (3/3)

• **What about jumps (j and jal)?**
  — Jumps require **absolute address**.
  — So, forward or not, still can’t generate machine instruction without knowing the position of instructions in memory.
• **What about references to data?**
  — **la** gets broken up into **lui** and **ori**
  — These will require the full 32-bit address of the data.
• These can’t be determined yet, so we create two tables...

Symbol Table

• List of “items” in this file that may be used by other files.
• **What are they?**
  — Labels: function calling
  — Data: anything in the .data section; variables which may be accessed across files
Relocation Table
• List of “items” this file needs the address of later.
• What are they?
  – Any label jumped to: J or jal
    • internal
    • external (including lib files)
  – Any piece of data that references an address
    • such as the la instruction

Object File Format
• object file header: size and position of the other pieces of the object file
• text segment: the machine code
• data segment: binary representation of the data in the source file
• relocation information: identifies lines of code that need to be “handled”
• symbol table: list of this file’s labels and data that can be referenced
• debugging information
  • A standard format is ELF (except MS, Apple)
    http://www.skyfree.org/linux/references/ELF_Format.pdf

Agenda
• I and J Instruction Formats
• Administrivia
• The Assembler
• Break
  • Linking
  • Conclusion

Separate Compilation and Assembly
• No need to compile all code at once
• How to put pieces together?

Linker Stitches Files Together

FIGURE B.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file. Copyright © 2009 Elsevier, Inc. All rights reserved.

FIGURE B.3.1 The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files. Copyright © 2009 Elsevier, Inc. All rights reserved.
Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Often a slower than compiling
  — all the machine code files must be read into memory and linked together

“And in Conclusion, ...”

- Everything is a (binary) number in a computer
  — Instructions are data; stored program concept
  — Different addressing schemes are needed due to the limitations of the I-Format and J-Format
- Assemblers support pseudoinstructions for the assembly language programmer, keep track of labels and data locations for linker.
- Linkers allow separate compilation and assembly of modules

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Disassembling Example (1/7)

- Here are six machine language instructions in hexadecimal:
  - 00001025_{hex}
  - 0005402A_{hex}
  - 11000003_{hex}
  - 00441020_{hex}
  - 20A5FFFF_{hex}
  - 08100001_{hex}
- Let the first instruction be at address 4,194,304_{ten} (0x00400000_{hex}).
- Next step: convert hex to binary

Disassembling Example (2/7)

- The six machine language instructions in binary:
  - 00000000000000000001000000100101
  - 00000000000001010100000000101010
  - 00010001000000000000000000000011
  - 00000000010001000001000000100000
  - 00100000101001011111111111111111
  - 00001000000100000000000000000001
- Look at opcode:
  - 0 means R-Format,
  - 2 or 3 mean J-Format,
  - otherwise I-Format.
- Next step: separation of fields

Disassembling Example (3/7)

- Select the opcode (first 6 bits) to determine the format:
  - Format:
    - R 00000000000000000001000000100101
    - R 00000000000000100100000000101010
    - I 00010001000000000000000000000011
    - R 00000000010001000001000000100000
    - I 00100000101001011111111111111111
    - J 00000000010001000000000000000001
- Look at opcode:
  - 0 means R-Format,
  - 2 or 3 mean J-Format,
  - otherwise I-Format.
- Next step: separation of fields
Disassembling Example

Fields separated based on format(opcode):

<table>
<thead>
<tr>
<th>R</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

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

Disassembling Example (5/7)

MIPS Assembly (Part 1):

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly instructions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>or $2, $0, $0</td>
</tr>
<tr>
<td>0x00400004</td>
<td>slt $8, $0, $5</td>
</tr>
<tr>
<td>0x00400008</td>
<td>beq $8, $0, 3</td>
</tr>
<tr>
<td>0x0040000c</td>
<td>add $2, $2, $4</td>
</tr>
<tr>
<td>0x00400010</td>
<td>add $5, $5, $1</td>
</tr>
<tr>
<td>0x00400014</td>
<td>j 0x100001</td>
</tr>
<tr>
<td>0x00400018</td>
<td></td>
</tr>
</tbody>
</table>

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

Disassembling Example (6/7)

MIPS Assembly (Part 2): 

<table>
<thead>
<tr>
<th>before hex</th>
<th>after C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001025</td>
<td>$v0: product $a0: multiplicand $a1: multiplier</td>
</tr>
<tr>
<td>0x0054022</td>
<td>product = 0;</td>
</tr>
<tr>
<td>110000003</td>
<td>while (multiplier &gt; 0) {</td>
</tr>
<tr>
<td>00441020</td>
<td>product += multiplicand;</td>
</tr>
<tr>
<td>20A5FFFF</td>
<td>multiplier -= 1;</td>
</tr>
<tr>
<td>08100001</td>
<td>}</td>
</tr>
</tbody>
</table>

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

Disassembling Example (7/7)

After C code

<table>
<thead>
<tr>
<th>before hex</th>
<th>Demonstrated Big 61C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001025</td>
<td>Idea: Instructions are just numbers, code is treated like data</td>
</tr>
<tr>
<td>0x0054022</td>
<td>or $v0, $0, $0</td>
</tr>
<tr>
<td>110000003</td>
<td>Loop: slt $t0, $0, $a1</td>
</tr>
<tr>
<td>00441020</td>
<td>beq $t0, $0, Exit</td>
</tr>
<tr>
<td>20A5FFFF</td>
<td>add $v0, $v0, $a0</td>
</tr>
<tr>
<td>08100001</td>
<td>addi $a1, $a1, -1</td>
</tr>
<tr>
<td>j Loop</td>
<td>Exit:</td>
</tr>
</tbody>
</table>

or $v0, $0, $0 

Loop: slt $t0, $0, $a1 

beq $t0, $0, Exit 

add $v0, $v0, $a0 

addi $a1, $a1, -1 

j Loop 

Exit: |