CS3350B Computer Architecture
MIPS Procedures and Compilation

Marc Moreno Maza

http://www.csd.uwo.ca/~moreno/cs3350_moreno/index.html
Department of Computer Science
University of Western Ontario, Canada

Tuesday January 30, 2018
Revisit: Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
Revisit: Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction
    - \[ \text{op} \quad \text{address} \]
    - 6 bits 26 bits
- (Pseudo)Direct jump addressing
  - Target address = \( \text{PC}_{31...28} \times (\text{address} \times 4) \)
Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

Loop: sll $t1, $s3, 2  
     add $t1, $t1, $s6  
     lw $t0, 0($t1)  
     bne $t0, $s5, Exit  
     addi $s3, $s3, 1  
     j Loop  
Exit: ...
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example
  
  ```
  beq $s0, $s1, L1
  ↓
  bne $s0, $s1, L2
  j L1
  ```
  
  L2: ...

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  lui rt, constant
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

```
lui $s0, 61  
ori $s0, $s0, 2304
```

<table>
<thead>
<tr>
<th>lui $s0, 61</th>
<th>ori $s0, $s0, 2304</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0111 1101 0000 0000 0000 0000</td>
<td>0000 0000 0111 1101 0000 1001 0000 0000</td>
</tr>
</tbody>
</table>
MIPS Organization So Far
Procedure Calling

Steps required

1. Place parameters in a place where the procedure can access them
2. Transfer control to the procedure
3. Acquire storage needed for the procedure
4. Perform procedure’s operations
5. Place result in a place where the calling procedure can access it
6. Return to control to the point of origin
Register Usage

- $a0 - $a3: arguments (reg’s 4 - 7)
- $v0, $v1: result values (reg’s 2 and 3)
- $t0 - $t9: temporaries
  - Can be overwritten by callee
- $s0 - $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)
Procedure Call Instructions

- Procedure call: **jump and link**
  - `jal ProcedureLabel`
    - Address of following instruction put in $ra
    - Jumps to target address

- Procedure return: **jump register**
  - `jr $ra`
    - Copies $ra to program counter
    - Can also be used for computed jumps
      - e.g., for case/switch statements
Leaf Procedure Example

- C code:
  ```c
  int leaf_example (int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
  }
  ```
  - Arguments $g, ..., j$ in $a0, ..., a3$
  - $f$ in $s0$ (hence, need to save $s0$ on stack)
  - Result in $v0$
# Leaf Procedure Example

- **MIPS code:**

<table>
<thead>
<tr>
<th>MIPS Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -4</td>
<td>Save $s0 on stack</td>
</tr>
<tr>
<td>sw $s0, 0($sp)</td>
<td></td>
</tr>
<tr>
<td>add $t0, $a0, $a1</td>
<td>Procedure body</td>
</tr>
<tr>
<td>add $t1, $a2, $a3</td>
<td></td>
</tr>
<tr>
<td>sub $s0, $t0, $t1</td>
<td></td>
</tr>
<tr>
<td>add $v0, $s0, $zero</td>
<td>Result</td>
</tr>
<tr>
<td>lw $s0, 0($sp)</td>
<td>Restore $s0</td>
</tr>
<tr>
<td>addi $sp, $sp, 4</td>
<td></td>
</tr>
<tr>
<td>jr $ra</td>
<td>Return</td>
</tr>
</tbody>
</table>
Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, ...
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings
String Copy Example

- C code ( naïve):
  - Null-terminated string
    ```c
    void strcpy (char x[], char y[]) {
        int i = 0;
        while (((x[i]=y[i])!='\0')
            i++;
    }
    ```
  - Addresses of x, y in $a0, $a1
  - i in $s0
String Copy Example

- **MIPS code:**

```
strcpy:
    addi $sp, $sp, -4  # adjust stack for 1 item
    sw $s0, 0($sp)    # save $s0

    add $s0, $zero, $zero  # i = 0

L1:     add $t1, $s0, $a1  # addr of y[i] in $t1
        lbu $t2, 0($t1)  # $t2 = y[i]

    add $t3, $s0, $a0  # addr of x[i] in $t3
    sb $t2, 0($t3)    # x[i] = y[i]

    beq $t2, $zero, L2  # exit loop if y[i] == 0

    addi $s0, $s0, 1  # i = i + 1
    j L1               # next iteration of loop

L2:     lw $s0, 0($sp)  # restore saved $s0
    addi $sp, $sp, 4  # pop 1 item from stack

    jr $ra            # and return
```
Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call
Non-Leaf Procedure Example

- C code:

```c
int fact (int n) {
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

- Argument n in $a0
- Result in $v0
Non-Leaf Procedure Example

- MIPS code:

<table>
<thead>
<tr>
<th>fact:</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -8</td>
</tr>
<tr>
<td>sw $ra, 4($sp)</td>
</tr>
<tr>
<td>sw $a0, 0($sp)</td>
</tr>
<tr>
<td>slti $t0, $a0, 1</td>
</tr>
<tr>
<td>beq $t0, $zero, L1</td>
</tr>
<tr>
<td>addi $v0, $zero, 1</td>
</tr>
<tr>
<td>addi $sp, $sp, 8</td>
</tr>
<tr>
<td>jr $ra</td>
</tr>
<tr>
<td>L1:</td>
</tr>
<tr>
<td>addi $a0, $a0, -1</td>
</tr>
<tr>
<td>jal fact</td>
</tr>
<tr>
<td>lw $a0, 0($sp)</td>
</tr>
<tr>
<td>lw $ra, 4($sp)</td>
</tr>
<tr>
<td>addi $sp, $sp, 8</td>
</tr>
<tr>
<td>mul $v0, $a0, $v0</td>
</tr>
<tr>
<td>jr $ra</td>
</tr>
</tbody>
</table>
Local Data on the Stack

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage
Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp$ initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - e.g., malloc in C, new in Java
- Stack: automatic storage
C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)
  ```c
  void swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
  }
  ```
  - v in $a0, k in $a1, temp in $t0
The Procedure Swap

<table>
<thead>
<tr>
<th>swap:</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll</td>
<td>sll $t1, $a1, 2</td>
<td>$t1 = k * 4</td>
</tr>
<tr>
<td>add</td>
<td>add $t1, $a0, $t1</td>
<td>$t1 = v + (k*4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(address of v[k])</td>
</tr>
<tr>
<td>lw</td>
<td>lw $t0, 0($t1)</td>
<td>$t0 (temp) = v[k]</td>
</tr>
<tr>
<td>lw</td>
<td>lw $t2, 4($t1)</td>
<td>$t2 = v[k+1]</td>
</tr>
<tr>
<td>sw</td>
<td>sw $t2, 0($t1)</td>
<td>v[k] = $t2 (v[k+1])</td>
</tr>
<tr>
<td>sw</td>
<td>sw $t0, 4($t1)</td>
<td>v[k+1] = $t0 (temp)</td>
</tr>
<tr>
<td>jr</td>
<td>jr $ra</td>
<td>return to calling routine</td>
</tr>
</tbody>
</table>
The Sort Procedure in C

- Non-leaf (calls swap)
  ```c
  void sort (int v[], int n) {
    int i, j;
    for (i = 0; i < n; i += 1) {
      for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1)
        swap(v, j);
    }
  }
  ```

- v in $a0, k in $a1, i in $s0, j in $s1
The Full Procedure

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -20</td>
<td># make room on stack for 5 registers</td>
</tr>
<tr>
<td>sw $ra, 16($sp)</td>
<td># save $ra on stack</td>
</tr>
<tr>
<td>sw $s3,12($sp)</td>
<td># save $s3 on stack</td>
</tr>
<tr>
<td>sw $s2, 8($sp)</td>
<td># save $s2 on stack</td>
</tr>
<tr>
<td>sw $s1, 4($sp)</td>
<td># save $s1 on stack</td>
</tr>
<tr>
<td>sw $s0, 0($sp)</td>
<td># save $s0 on stack</td>
</tr>
<tr>
<td>...</td>
<td># procedure body</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>lw $s0, 0($sp)</td>
<td># restore $s0 from stack</td>
</tr>
<tr>
<td>lw $s1, 4($sp)</td>
<td># restore $s1 from stack</td>
</tr>
<tr>
<td>lw $s2, 8($sp)</td>
<td># restore $s2 from stack</td>
</tr>
<tr>
<td>lw $s3,12($sp)</td>
<td># restore $s3 from stack</td>
</tr>
<tr>
<td>lw $ra,16($sp)</td>
<td># restore $ra from stack</td>
</tr>
<tr>
<td>addi $sp, $sp, 20</td>
<td># restore stack pointer</td>
</tr>
<tr>
<td>jr $ra</td>
<td># return to calling routine</td>
</tr>
</tbody>
</table>
### The Procedure Body

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $s2, $a0</td>
<td># save $a0 into $s2</td>
</tr>
<tr>
<td>move $s3, $a1</td>
<td># save $a1 into $s3</td>
</tr>
<tr>
<td>move $s0, $zero</td>
<td># i = 0</td>
</tr>
<tr>
<td>for1tst: slt $t0, $s0, $s3</td>
<td># $t0 = 0 if $s0 $geq $s3 (i $geq n)</td>
</tr>
<tr>
<td>for2tst: beq $t0, $zero, exit1</td>
<td># go to exit1 if $s0 $geq $s3 (i $geq n)</td>
</tr>
<tr>
<td>addi $s1, $s0, -1</td>
<td># j = i - 1</td>
</tr>
<tr>
<td>slti $t0, $s1, 0</td>
<td># $t0 = 1 if $s1 &lt; 0 (j &lt; 0)</td>
</tr>
<tr>
<td>bne $t0, $zero, exit2</td>
<td># go to exit2 if $s1 &lt; 0 (j &lt; 0)</td>
</tr>
<tr>
<td>sll $t1, $s1, 2</td>
<td># $t1 = j * 4</td>
</tr>
<tr>
<td>add $t2, $s2, $t1</td>
<td># $t2 = v + (j * 4)</td>
</tr>
<tr>
<td>lw $t3, 0($t2)</td>
<td># $t3 = v[j]</td>
</tr>
<tr>
<td>lw $t4, 4($t2)</td>
<td># $t4 = v[j + 1]</td>
</tr>
<tr>
<td>slt $t0, $t4, $t3</td>
<td># $t0 = 0 if $t4 $geq $t3</td>
</tr>
<tr>
<td>beq $t0, $zero, exit2</td>
<td># go to exit2 if $t4 $geq $t3</td>
</tr>
<tr>
<td>move $a0, $s2</td>
<td># 1st param of swap is v (old $a0)</td>
</tr>
<tr>
<td>move $a1, $s1</td>
<td># 2nd param of swap is j</td>
</tr>
<tr>
<td>jal swap</td>
<td># call swap procedure</td>
</tr>
<tr>
<td>addi $s1, $s1, -1</td>
<td># j -= 1</td>
</tr>
<tr>
<td>j for2tst</td>
<td># jump to test of inner loop</td>
</tr>
<tr>
<td>exit2: addi $s0, $s0, 1</td>
<td># i += 1</td>
</tr>
<tr>
<td>j for1tst</td>
<td># jump to test of outer loop</td>
</tr>
</tbody>
</table>
Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination
  - `move $t0, $t1` → `add $t0, $zero, $t1`
  - `blt $t0, $t1, L` → `slt $at, $t0, $t1`
  - `bne $at, $zero, L`
- `$at` (register 1): assembler temporary
Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends on order of accesses
- Hardware support required
  - Atomic read/write memory operation
    - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions
Synchronization in MIPS

- **Load linked**: `ll rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
  - Returns 1 in `rt`
- **Store conditional**: `sc rt, offset(rs)`
  - Fails if location is changed
  - Returns 0 in `rt`

- Example: atomic swap (to test/set lock variable)
  ```assembly
  try: add $t0, $zero, $s4  # copy exchange value
  ll  $t1, 0($s1) # load linked
  sc  $t0, 0($s1) # store conditional
  beq $t0, $zero, try # branch store fails
  add $s4, $zero, $t1 # put load value in $s4
  ```
Translation and Startup

Many compilers produce object modules directly.

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory

Static linking
Producing an Object Module

- **Assembler** (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - **Header**: described contents of object module
  - **Text segment**: translated instructions
  - **Static data segment**: data allocated for the life of the program
  - **Relocation info**: for contents that depend on absolute location of loaded program
  - **Symbol table**: global definitions and external refs
  - **Debug info**: for associating with source code
Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space
Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, ... and calls main
     - When main returns, do exit syscall
Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

a. First call to DLL routine
b. Subsequent calls to DLL routine
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

- Relative Performance
- Instruction count
- Clock Cycles
- CPI
Effect of Language and Algorithm

![Bubblesort Relative Performance](chart1)

![Quicksort Relative Performance](chart2)

![Quicksort vs. Bubblesort Speedup](chart3)
Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!
Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Takeaway: MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions - 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits
- Good design demands good compromises
  - 3 basic instruction formats
- Smaller is faster
  - limited instruction set
  - limited number (32) of registers in register file
  - limited number (5) of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

Question: How Can We Make It Even Faster?
Aside: ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

<table>
<thead>
<tr>
<th>Feature</th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>$15 \times 32$-bit</td>
<td>$31 \times 32$-bit</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
Aside: The Intel x86 ISA

Evolution with backward compatibility

- **8080 (1974):** 8-bit microprocessor
  - Accumulator, plus 3 index-register pairs
- **8086 (1978):** 16-bit extension to 8080
  - Complex instruction set (CISC)
- **8087 (1980):** floating-point coprocessor
  - Adds FP instructions and register stack
- **80286 (1982):** 24-bit addresses, MMU
  - Segmented memory mapping and protection
- **80386 (1985):** 32-bit extension (now IA-32)
  - Additional addressing modes and operations
  - Paged memory mapping as well as segments
Aside: The Intel x86 ISA

Further evolution...

- i486 (1989): pipelined, on-chip caches and FPU
  - Compatible competitors: AMD, Cyrix, ...
- Pentium (1993): superscalar, 64-bit datapath
  - Later versions added MMX (Multi-Media eXtension) instructions
  - The infamous FDIV bug
  - New microarchitecture (see Colwell, The Pentium Chronicles)
- Pentium III (1999)
  - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions
Aside: The Intel x86 ISA

And further...

- **AMD64 (2003):** extended architecture to 64 bits
- EM64T - Extended Memory 64 Technology (2004)
  - AMD64 adopted by Intel (with refinements)
  - Added SSE3 instructions
- Intel Core (2006)
  - Added SSE4 instructions, virtual machine support
- **AMD64 (announced 2007):** SSE5 instructions
  - Intel declined to follow, instead...
- Advanced Vector Extension (announced 2008)
  - Longer SSE registers, more instructions

If Intel didn’t extend with compatibility, its competitors would!

- Technical elegance ≠ market success