CS 230 – Introduction to Computers and Computer Systems

Module 2 – Assembly Language
Overview

- Assembly language: MIPS
  - arithmetic operations
  - data movement
  - conditional execution
  - subroutines
Machine Code

- Binary code – comprised of 0s and 1s
- “Direct” execution by processor
- Program grouped into instructions
  - fixed vs. variable length
  - operation code (opcode) + operands
  - instructions control processor
    - opcode designates operation
    - operands designate data
Assembly Language

- Human-readable “programming language”
  - very simple compared to, e.g., Racket, Python
- Almost direct mapping to machine code, except:
  - labels
  - data declaration
  - pseudo instructions
- Assembled into machine code
Instruction Set

- Repertoire of instructions
- Different processors have different sets
  - many commonalities
    - mathematical
    - memory access
    - control flow
Turing Completeness

- Well-defined theoretical concept
- Fundamental capabilities of instruction set
- Minimum requirements
  - while or if/goto & change memory locations
- Typical programming languages / instruction sets are Turing complete
  - Data description languages (e.g. XML) are not
MIPS Architecture

- MIPS: Microprocessor without Interlocked Pipeline Stages
  - details later (Module 3)

- Multiple revisions, systems, and compilers
  - not just a single standard MIPS
MIPS Assembly Language

- Each instruction takes 32 bits
  - 4 bytes
  - 1 word
- Arithmetic instructions operate on registers
  - separate memory load and store
  - 32 registers available
- Instructions have up to 3 operands
  - 1\(^{st}\) is destination, 2\(^{nd}\) and 3\(^{rd}\) are sources
  - same register can be source and destination
Special Registers

- **PC** – program counter
  current location (byte address) in machine code
  - incremented by 4 for each instruction
- **$0** – constant 0
- Register conventions (more later)
  - **$29** – stack pointer (**$sp**)  
  - **$30** – frame pointer (**$fp**)  
  - **$31** – return address (**$ra**)
MIPS Assembly Language

- Two addressing modes
  - register: operands in registers
  - immediate: one operand is a 16-bit constant
- Memory: 32-bit address space
  - relative & absolute memory load/store
  - relative & absolute branch instructions

- CS 230: subset of actual MIPS language
Addition

add $3, $2, $1

- add (contents of) register '1' and '2'
- place result in register '3'
- general notation: add $d, $s, $t
- move (contents of ) register '1' to '2': add $2, $1, $0

sub $d, $s, $t

- subtract register 't' from 's'
- place result in register 'd'
Immediate Addition

\texttt{addi \$t, \$s, i}

- add register 's' and value \( i \)
- value \( i \) can be negative – size/range?
- place result in register 't'
- use to initialize registers: \texttt{addi \$t, \$0, i}
  - set register 't' to value \( i \)
Multiplication

- special registers *hi, lo*

**mult** $s, t$
  - multiply registers 's' and 't'
  - place result in *hi:lo*

**div** $s, t$
  - divide register 's' by 't'
  - place result in *lo*, remainder in *hi*
Multiplication Results

mfhi $d
  • copy contents of hi to $d
mflo $d
  • copy contents of lo to $d
Example

- Compute average of three numbers
  - values in $3, $4, $5
  - result in $2
Example: Average

add $2, $3, $4
add $2, $2, $5
addi $1, $0, 3
div $2, $1
mflo $2
Example

- Compute average of three numbers
- Second version: reduce register usage
  - values in $3, $4, $5
  - result in $3
  - but original value $3 overwritten
Example: Average Again

add $3, $3, $4
add $3, $3, $5
addi $1, $0, 3
div $3, $1
mflo $3
Example

- Compute $(a+b) - (c+d)$
  - $a, b, c, d$ in $1, 2, 3, 4$ – preserve values
  - result in $5$
- Need to compute temporary values
  - use $5$ and $6$
Example: Subtract Sums

add $5, $1, $2
add $6, $3, $4
sub $5, $5, $6
Example: Subtract Sums

- Do $a + b - c - d$ instead
  - 1 fewer registers

add $5, 1, 2$
sub $5, 5, 3$
sub $5, 5, 4$
Assembler Invocation in CS230

- Assembler is \( /u/cs230/pub/binasm \)
- Input is standard input stream
- Output (binary file) on standard output
- Assume `bash` is used as shell
  - if necessary, invoke `bash` explicitly
    \( /u/cs230/pub/binasm < \text{ifile} > \text{ofile} \)
MIPS Emulator in CS230

- Several frontends in /u/cs230/pub
  - twoints <mips-file>
    enter two integer numbers – stored in $1, $2
  - array <mips-file>
    enter array of integer numbers – start stored in $1, length in $2
  - noargs <mips-file>
    run code immediately without expecting predefined user input
- Argument <mips-file> is binary code
  - output from assembler
Debugging

- Difficult
  - use output – next topic in lecture
  - or terminate (jr $31) to study register setting
    - verify program step by step

- General techniques
  - analyze log output
  - controlled step-by-step execution
    - some kind of virtual environment needed
  - verify assertions
Conditional Execution

- Essential feature of von Neumann computers!
- Computation dependent on intermediate results
  - Otherwise only linear number crunching
Conditional Branch

beq $s, $t, i
  • compare registers 's' and 't'
  • if equal, skip $i$ instructions
  • $i$ can be negative

bne $s, $t, i
  • compare registers 's' and 't'
  • if not equal, skip $i$ instructions
  • $i$ can be negative
Conditional Branch

- Reference sheet notation
  \[ \text{beq } s, t, i \]
  - \[ \text{if ( } s == t \text{ ) } PC += i \times 4 \]
- Notes
  - \[ PC = PC + i \times 4 \]
  - word length is 4
  - in addition to regular PC increment
  - *relative* (to current PC) branch
# Example – Loop

<table>
<thead>
<tr>
<th>Example PC Value</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>addi $2, $0, 1</td>
</tr>
<tr>
<td>04</td>
<td>addi $1, $0, 10</td>
</tr>
<tr>
<td>08</td>
<td>mult $2, $1</td>
</tr>
<tr>
<td>0C</td>
<td>mflo $2</td>
</tr>
<tr>
<td>10</td>
<td>addi $1, $1, -1</td>
</tr>
<tr>
<td>14</td>
<td>bne $1, $0, -4</td>
</tr>
</tbody>
</table>
Branch Labels

- Leave computation of branch offset to assembler
  - Now we don’t have to change a number when we change the loop body

```assembly
addi $1, $0, 10

loop:
    ; do something
    addi $1, $1, -1
    bne $1, $0, loop
```
Conditional Execution

; compute something
beq $1, $2, eqcase
; do something for $1 != $2
beq $0, $0, final

eqcase:
; do something for $1 == $2

final:
; continue with program
Label Naming

- Readable and intuitive vs. unique!
- Name scope?
  - source file for now
  - internal vs. external – more later (Module 4)
- Manual vs. automatic generation
Set Less Than

\texttt{slt} \ $d, \ $s, \ $t$

- compare register 't' and 's'
- \( s < t \Rightarrow d = 1 \)
- \( s \geq t \Rightarrow d = 0 \)
Constants

- Can be done with \texttt{addi}
  - but limited value range (16 bits)
- Alternative: \texttt{lis} instruction
  - load next word (32-bits) into register and skip
  - Example: load value 0xA3257CE2 into register $4
    \begin{verbatim}
    lis $4
    \end{verbatim}
    \begin{verbatim}
    .word 0xA3257CE2
    \end{verbatim}
Memory Model

- Memory region
  - $2^{32}$ bytes
- Byte addressable
  - $2^{32} = 4294967296$ addresses
- Word aligned
  - can only access whole words
    - only $2^{30} = 1073741824$ valid words to load
  - also called “word referenced”
Memory Access

\texttt{lw} \ $t, \ i(\$s)$

- load word from location 's' + i into 't'
- 's' + i must be word-aligned

\texttt{sw} \ $t, \ i(\$s)$

- store word from 't' into location 's' + i
- 's' + i must be word-aligned
Memory Addresses

- Make sure to distinguish between
  - register number: between 0 and 31
  - register contents: expressed as, e.g., $20
  - memory address in \texttt{lw} or \texttt{sw} instruction
  - memory contents (accessed via \texttt{lw} or \texttt{sw})
- From module 1:
  - bits are in the eye of the beholder
Structures

- High-level structures
  - Like you might see in C

```c
struct BirthRecord {
    int year;
    int month;
    int day;
    char* city;
}
```
Structures

- Assume birth record is stored at address $1
  - access year: \texttt{lw $t, 0($1)}
  - access month: \texttt{lw $t, 4($1)}
  - access day: \texttt{lw $t, 8($1)}
  - access start of city: \texttt{lw $t, 12($1)}
Memory Access Example

- I know there is a birth record at address 0x44, how do I get the month?

  addi $1, $0, 0x44
  lw $2, 4($1)

or

  addi $1, $0, 0x48
  lw $2, 0($1)

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40</td>
<td>0x00FF00FF</td>
</tr>
<tr>
<td>0x44</td>
<td>0x000007D0</td>
</tr>
<tr>
<td>0x48</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>0x4C</td>
<td>0x00000013</td>
</tr>
<tr>
<td>0x50</td>
<td>0x57617465</td>
</tr>
<tr>
<td>0x54</td>
<td>0x726C6F6F</td>
</tr>
<tr>
<td>0x54</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
Memory Access Example

- What if I want the year and the month?

\[
\text{addi } \$1, \$0, 0x44 \\
\text{lw } \$2, 0(\$1) \\
\text{lw } \$3, 4(\$1)
\]

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40</td>
<td>0x00FF00FF</td>
</tr>
<tr>
<td>0x44</td>
<td>0x000007D0</td>
</tr>
<tr>
<td>0x48</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>0x4C</td>
<td>0x00000013</td>
</tr>
<tr>
<td>0x50</td>
<td>0x57617465</td>
</tr>
<tr>
<td>0x54</td>
<td>0x726C6F6F</td>
</tr>
<tr>
<td>0x54</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
Memory Access Example

- How do I set the day to the 15\textsuperscript{th}?

\begin{verbatim}
addi $1, $0, 0x44
addi $2, $0, 15
sw $2, 8($1)
\end{verbatim}

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40</td>
<td>0x00FF00FF</td>
</tr>
<tr>
<td>0x44</td>
<td>0x000007D0</td>
</tr>
<tr>
<td>0x48</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>0x4C</td>
<td>0x0000000F</td>
</tr>
<tr>
<td>0x50</td>
<td>0x57617465</td>
</tr>
<tr>
<td>0x54</td>
<td>0x726C6F6F</td>
</tr>
<tr>
<td>0x54</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
Low Level Errors

- Illegal instruction
- Assignment to read-only register
- Integer division by 0
- Alignment violation
- Memory protection violation
- Etc.
  - usually result in exception and termination
Assembly File

- Assembly instructions
- Label declarations
- Data definitions (.word)
- Comments – start with semicolon
Assembly File – Example Format

- 3 Columns: label, instruction / data, comment
- Example

  loop:      lis $1
             .word 0x12
  lis $3
             .word data  ; can use label
  data:     .word 0x20  ; 0x20 = 32
             .word 32
Input / Output

- Input memory address FFFF0004\textsubscript{16}
  - load from this address retrieves keyboard input one character at a time
  - “Magic” address, each read has a new value

- Output memory address FFFF000C\textsubscript{16}
  - bytes written to this address appear on screen
  - ASCII encoding, only the least significant 7 bits

- Only use actual keyboard input
  - Input redirection may not work
I/O Example

lis $1
.word 0xFFFF0004
lis $2
.word 0xFFFF000C
addi $3, $0, 0x1B ; ESC character
loop: lw $4, 0($1)
sw $4, 0($2)
sw $4, 0($2)
bne $3, $4, loop
jr $31
Notes

- Interactive program
  - control execution through I/O

- I/O ports mapped to memory
  - here somewhat simplified
  - but not too far from reality
Iterating through String

lis $1
.word 0xffff000c
lis $2
.word print

loop: lw $3, 0($2)
beq $3, $0, end
sw $3, 0($1)
addi $2, $2, 4
beq $0, $0, loop
Iterating through String

end:    jr $31

print:  .word 0x43   ; C
         .word 0x53   ; S
         .word 0x32   ; 2
         .word 0x33   ; 3
         .word 0x30   ; 0
         .word 0x0A   ; LF
         .word 0x00   ; NUL
Array

- Use memory range as data area
  - can iterate through addresses
- Direct access and iteration possible
- “Raw” array
  - no built-in boundary checking
- Memory allocation/management
  - details later
Case Study: Selection Sort

- Simple sorting algorithm
  - not necessarily best or fastest, but simple
  - sweep through array
    - find smallest element and move to front
    - then restart sweep at next position
Selection Sort – Python

- `def SelectionSort(A, length):
  for i in range(0, length-1):
    minpos = i
    minimum = A[i]
    for j in range(i+1, length):
      if A[j] < minimum:
        minpos = j
        minimum = A[j]
    temp = A[i]
    A[minpos] = temp`
Selection Sort – MIPS

- Use `length` and `i` (simplify assembly)
  - in python: `range (a, l): a, a+1, a+2,..., a+l-1`
- Start of array in $1$
- Length of array in $2$
- Local variables:
  - end of array in $8$
  - i in $9$, j in $12$
  - minpos in $10$, minimum in $11$
  - temps in $13$, $14$
Selection Sort – MIPS

- Set up array end for easier looping

  ```mips
  addi $8, $0, 4          ; set up array end
  mult $2, $8
  mflo $8
  add $8, $8, $1
  ```
Selection Sort – Main Program

addi $9, $1, 0 ; initialize $9

oloop: addi $10, $9, 0 ; set minpos $10
lw $11, 0($10) ; set minimum $11
addi $12, $9, 0 ; initialize $12

iloop: lw $13, 0($12) ; set up compare value
slt $14, $13, $11
beq $14, $0, notmin
addi $11, $13, 0 ; set new minimum
addi $10, $12, 0 ; set new minpos

notmin: addi $12, $12, 4
bne $12, $8, iloop ; loop over $12
lw $14, 0($9) ; load front value
sw $14, 0($10) ; store at minpos
sw $11, 0($9) ; store min in front
addi $9, $9, 4
bne $9, $8, oloop ; loop over $9
Selection Sort – Lessons Learned

- Manually keeping track of registers is difficult!
- Otherwise structure similar to python program
Subroutine

- Important technique to modularize programs
  - separation of concerns, smaller coding units
  - reuse of code – call from anywhere

- Challenges
  - call/return – how to redirect execution?
    - how do I get back to where I came from?
  - nested call/return, recursion
  - argument and result passing
Subroutine

jal x

- copy (current PC + 4) to $31
- set PC to x (long immediate)

jr $s

- set PC to $s
- usually jr $31
- convention: register $31 holds return address
Subroutine

- Subroutines can call subroutines
  - including themselves - recursion
- How about registers that are in use?
  - across subroutine calls
  - registers ≠ local variables
- Save/restore current *execution context*
  - set of register values
  - save where?
Example: Print String

- Print NUL-terminated string
- Use register conventions
  - $4...$7 for arguments
  - $8...$15 for local variables
- Simple
  - do not need the stack yet
Example: Print String

; pr_str: prints out a null-terminated string
; input:   $4
; locals:  $8, $9, $10

pr_str:  lis $8                  ; set $8 for output
        .word 0xfffffffff000c
        addi $9, $4, 0         ; copy value into $9
loop:   lw  $10, 0($9)          ; load character
        beq  $10, $0, end     ; NUL? -> end
        sw   $10, 0($8)       ; output character
        addi $9, $9, 4       ; increment $9
        beq  $0, $0, loop    ; loop
end:    addi $10, $0, 0xA       ; print LF
        sw   $10, 0($8)
        jr   $31              ; return
Stack

- Define memory area as *stack*
  - last-in first-out queue
  - convention: stack grows downward in memory
- Address “bottom” by *stack pointer* register
  - convention: register $29 on MIPS
  - convention: $29 points to last used word
- **HOWEVER**: in our MIPS simulator...
  - use $30 instead
Stack Save/Restore

- **Save register on stack – push**
  - decrement stack pointer to make room
  - copy value to stack pointer memory location
    
    addi $30, $30, -4
    sw $x, 0($30)

- **Restore register from stack – pop**
  - copy value from stack pointer memory location
  - increment stack pointer to free up space
    
    lw $x, 0($30)
    addi $30, $30, 4
Argument Passing

- Need to pass arguments and result
- Can use registers and/or stack
  - need to agree between caller and callee
  - MIPS convention
    - first 4 arguments in registers
    - remainder on the stack
Register Conventions

- Which registers to save?
- Which registers to use for arguments/results?
- MIPS
  - $1 – assembler temporary
  - $2, $3 – function results and expressions
  - $4 - $7 – arguments
  - $8 - $15, $24, $25 – temporary
  - $16 - $23 – saved temporary
  - $26 - $27 – OS kernel
Who Saves What?

- Caller vs. callee saves registers
  - typical: more callers than callees
    - callee saves registers produces less code
    - save registers only if necessary
  - MIPS convention
    - callee saves $16-$23
      - if you want to use one of these: save it first
    - caller must save $8 - $15, $24, $25 and others
      - if you call another function: save these first or lose them
    - clean up the stack before jr $31
Local Variables

- Default: make room on stack
- Usually no limits
  - number of local variables
  - size of local variables
- “Automatic” variables
  - memory reserved on routine entry
  - memory released on routine exit
Stack Frame

- Stack pointer: bottom of stack, first free field
- Stack pointer might vary during routine
  - set up stack for subroutine call
- At routine entry, frame pointer is set
  - e.g. to stack pointer at that time
- Frame pointer provides a constant base to
  - access local variables and arguments
- use offset in `lw $t, i($fp)` instruction
Hardware Support

- Often support from hardware / assembler
  - reserving stack area, save registers
  - register windows (not on MIPS)
    - multiple instances (windows) of callee-saved registers
    - “save” -> use next window
    - fast but limited to fixed small number of windows
    - e.g. Sparc: overlapping windows
      - 8x input, 8x local, 8x output
      - shift by 16, i.e., output becomes input
Example: Print Integer

- Extract digits
  - check for negative number
  - use modulo 10 and remainder to get digits
  - convert digits to ASCII
- Use stack to reorder digits
Example: Print Integer – Part 1

; pr_int: prints out an integer value
; input:  $4
; locals: $8, $9, $10, $11

pr_int:  lis $8             ; set up $8 for output
         .word 0xffffffffc
        addi $9, $4, 0       ; set up value $9
        addi $11, $0, 0     ; set up counter $11
        slt $10, $9, $0     ; check for negative
        beq $10, $0, comp
        addi $10, $0, 0x2D  ; print minus sign
        sw $10, 0($8)
        sub $9, $0, $9      ; make $9 positive
Example: Print Integer – Part 2

comp:     addi $11, $11, 1  ; increment counter
         addi $10, $0, 10
         div $9, $10       ; divide by 10
         mfhi $10
         addi $30, $30, -4 ; remainder on stack
         sw $10, 0($30)
         mflo $9            ; result to $9
         bne $9, $0, comp   ; restart loop
Example: Print Integer – Part 3

output:   lw $10, 0($30) ; start from stack
          addi $30, $30, 4
          addi $10, $10, 0x30 ; convert to ASCII
          sw $10, 0($8) ; output
          addi $11, $11, -1 ; decrement counter
          bne $11, $0, output ; restart loop
          addi $10, $0, 0xA ; print LF
          sw $10, 0($8)
          jr $31 ; return
Recursion

- Mathematical programming technique
- Divide larger problem into smaller similar ones
- Recursion mandates stack
  - need memory area per subroutine

- Recursion fundamentally equivalent to
  - loop
  - conditional branch
Example: Factorial

- Assume input is in $4
- Output to $2
- Use recursion
  - save only those registers necessary
  - in particular, $2 is not saved during recursion
  - only modified on outbound path
Example: Factorial

```assembly
fac:    addi $2, $0, 1 ; basic result
        slt $3, $0, $4 ; $4 > 0 ?
        beq $3, $0, end ; else: finish
        addi $30, $30, -8 ; save $31 & $4
        sw $31, 4($30)
        sw $4, 0($30)
        addi $4, $4, -1 ; decrement $4
        jal fac ; recursion
r:     lw $31, 4($30) ; restore $31 & $4
        lw $4, 0($30)
        addi $30, $30, 8
        mult $2, $4 ; multiply
        mflo $2 ; store result
end:    jr $31 ; return
```
Case Study: Fibonacci

- Defined as:
  - $f(n) = f(n-1) + f(n-2)$
  - $f(1) = 1$
  - $f(0) = 0$

- In Python:

```python
def fib(n):
    if n < 2:  # base case
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```
Fibonacci – MIPS

; input:  $4
; output: $2
; locals:  $8

fib:    addi $2, $4, 0     ; default output = input
        addi $8, $0, 1
        slt $8, $8, $4     ; 1 < input
        beq $8, $0, end    ; else: finish

        addi $30, $30, -8  ; save return & input
        sw $31, 4($30)
        sw $4, 0($30)
Fibonacci – MIPS

```assembly
addi $4, $4, -1  ; compute f(n-1)
jal fib           ; result in $2
addi $30, $30, -4 ; store result on stack
sw $2, 0($30)
addi $4, $4, -1   ; compute f(n-2)
jal fib           ; result in $2
lw $8, 0($30)     ; restore f(n-1) as $8
addi $30, $30, 4  
add $2, $2, $8    ; add f(n-1)+f(n-2)

lw $31, 4($30)   ; restore return & input
lw $4, 0($30)
addi $30, $30, 8

end:   jr $31     ; return
```
Notes

- Make room on stack for several variables
- Follow register conventions
  - save input variable, because it is changed
  - save return address as usual