Running Code on a Microprocessor

- Function of the microprocessor is to fetch and execute instructions from memory.
- Microprocessor runs machine code – ones and zeros.
- Need tools to make high-level code recognizable to the processor.
- Compiler
  - Translate source code into assembly code.
  - Optimize the generated code.
- Assembler
  - Translate the assembly code into binary, objective code.
  - Resolve symbols in assembly.
- Linker
  - Integrate multiple binary objectives into a single binary executable.
- Loader
  - Load the binary executable of program into memory, and transfer the program control to the start of the program.

Assembly Language Programming

- Assembly language is a low level language.
- Programmer is the compiler.
- Each mnemonic translates directly into machine code.
- Good assembly code can provide maximum speed and minimum memory usage.
- Why do we learn machine-level programming?
  - Sometimes code must be written in assembly – register access.
  - Assembly is more efficient, if wisely coded.
  - Understand the rational and limits of C programming.
  - Understand how computer hardware works.

Assembly Language Programming

- Typical instruction types
  - Load and store: move data between registers and memory.
  - Arithmetic and logic: +, -, *, /, &, |, ^, and more.
  - Comparison: build conditions for branches.
  - Branch and jump: change sequential execution.
  - FP instructions: load/store, +, -, *, / and more for FP.
  - Miscellaneous: e.g. system calls.
- What to learn?
  - Machine model.
  - Registers and memory.
  - Memory addressing.
  - Instructions and operations.

Machine model

- Registers
  - Most frequently used data.
  - Vary fast access – operates at processor clock speed.

PPC Registers

- General purpose registers (integer registers)
  - 32, 32-bit registers – r0 to r31.
  - r0 is treated differently in some instructions.
- Floating point registers
  - 32, 64-bit registers – fr0 to fr31.
- Integer Exception Register (XER):
  - Indicates overflows and carry conditions for integer operations.
- Link register (LR):
  - Holds return address.
- Count register (CTR):
  - Holds a loop count; may be decremented automatically with special branches.

Memory Subsystems

- Memory operations
  - Read: accept address and return data.
  - Write: accept address and data, update memory.
- Cache and main memory
  - Cache: small, fast memory to hold hot inst/data (SRAM).
  - Main memory can be large but slow (DRAM).
- Cache speed must match processor speed.
Memory Address Space

- It is the addressability of the memory
  - Upper bound of memory that can be accessed by a program
  - The larger the space, the more bits in memory addresses
  - 32-bit address – accessibility to 4GB memory

What is
- Physical memory address space
- Virtual memory address space
- I/O addresses

Memory Problems

- Memory leak
  - When you allocate dynamic memory and don’t free it
- Segmentation fault
  - General Protection Fault in Windows
  - Try to access a restricted memory reserved for the OS
  - Try to write read only memory
  - Attempt to read instruction memory as data
  - Incorrect instruction format

Reduced Instruction Set Computer (RISC)

- Smaller and simpler instruction set
- Instructions take about the same amount of time to execute
- Same length instructions
- Simpler hardware
- Lower power consumption
- Primarily used for embedded systems
- Instruction mnemonic uniquely identifies the instruction

Memory Address Space

- static char myString[] = “Hello world”;
- int *myFunction()
  
  int i;
  char myVal;
  int *myMem = malloc(sizeof(int)*10);
  LCD_init();
  LCD_PutString(greeting);
  callFunction();
  return(myMem);

Moving From Complex Instructions Sets to Simple Instructions

- Early compilers were not available so it was convenient for programmers to have many instructions
  - One instruction to retrieve numbers add them and then store the result
  - Different instructions to load the numbers from registers/memory and store to register/memory and any combination
  - Orthogonality – each instruction was fine tuned to reduce overhead
- Increased complexity in CPU design including pipelining and parallelism required simpler and more uniform instructions
- This brought about the Reduced Instruction Set Computer or sometimes called load/store architecture

RISC Uses and Facts

- MIPS – Microprocessor without Interlocked Pipeline Stages (Stanford)
  - Focused mainly on pipeline – every instruction was required to be completed in one cycle (pipeline doesn’t need to stall – interlock free)
  - Complex instructions were eliminated such as multiply and divide
  - SGI workstations, Nintendo64, PlayStation, PSP, Cisco Routers – even Motorola/Freescale uses MIPS
- RISC project (Berkley) – SUN SPARC
- IBM POWER architecture (including the PowerPC)
- XBOX 360, Nintendo Revolution, Playstation 3
Load/Store Instructions

- Only load/store instructions can access memory
  - Various load/store instructions are used for different data size, data extension, etc. – consult the reference manual
- To access program variables from memory
  - Load variables from memory into registers
  - Perform arithmetic/logic operations
  - Store result back to memory
- Example – myVal = myVal + 100;
  lwz r5, 0x1000(r3) # r5 <- value at $r31 + 0x1000
  addi r5, r5, 100 # add 100 (0x64)
  stw r5, 0x1000(r3) # store back

Memory Addressing

- How to calculate effective memory address (EA)
  - Displacement EA = base register value + offset
  - Example: lwz r4, 0x1000(r3); EA = r3 + 0x1000
  - For absolute address: lwz r5, 0x1000(r0)
  - For register indirect: lwz r5, 0(r3)
- Register Indexed EA = base register value + index register value
  - Example: lwzx r5, r3, r4; EA = r3 + r4

Load/Store Instructions

- Assume $r3 = 0x2000 0000,
  mem(0x200000200) = 0x1234 5678
- Loading registers from memory – 3 sizes
  - Load byte
    - lbz r5, 0x200(r3) # r5 = 0x0000 0012
    - lhz r5, 0x201(r3) # r5 = 0x0000 0034
  - Load half word
    - lhz r5, 0x200(r3) # r5 = 0x0000 1234
    - lhz r5, 0x202(r3) # r5 = 0x0000 5678
  - Load word
    - lwz r5, 0x200(r3) # r5 = 0x1234 5678

Load/Store Instructions

- Assume $r3 = 0x2000 0000,
  mem(0x2000000200) = 0x1234 5678
- Loading registers from memory – 3 sizes
  - Load byte
    - lbz r5, 0x200(r3) # r5 = 0x0000 0012
    - lhz r5, 0x201(r3) # r5 = 0x0000 0034
  - Load half word
    - lhz r5, 0x200(r3) # r5 = 0x0000 1234
    - lhz r5, 0x202(r3) # r5 = 0x0000 5678
  - Load word
    - lwz r5, 0x200(r3) # r5 = 0x1234 5678

Load/Store Instructions

- Suppose mem($r3+0x1000) stores 0xFFFF 0000 (big-endian)
- How to fill the rest of memory when loading a byte or a short?
  - Zero extension – Fill with zero
  - Algebraic extension – Fill with the sign bit
  - Two examples
    - lhz r5, 0x1000(r3) ; r5 = 0x0000 0000
    - lia r5, 0x1000(r3) ; r5 = 0x0000 0000
  - 2 - zero, a – algebraic
  - What load instruction to use for m and n?
    - short m;
    - unsigned short n;

Load/Store Instructions

- Store instructions
  - Three data sizes – byte, half-word, word
  - Two addressing modes – displacement or register index
  - No extension issue
- Examples
  - stb r5, 0x1000(r3)
  - sth r5, 0x1000(r3)
  - stwx r5, r3, r4

Load/Store Instructions

- Store instructions
  - Three data sizes – byte, half-word, word
  - Two addressing modes – displacement or register index
  - No extension issue
- Examples
  - stb r5, 0x1000(r3)
  - sth r5, 0x1000(r3)
  - stwx r5, r3, r4

Load/Store Instructions

- Store instructions
  - Three data sizes – byte, half-word, word
  - Two addressing modes – displacement or register index
  - No extension issue
- Examples
  - stb r5, 0x1000(r3)
  - sth r5, 0x1000(r3)
  - stwx r5, r3, r4

Load/Store Instructions

- Store instructions
  - Three data sizes – byte, half-word, word
  - Two addressing modes – displacement or register index
  - No extension issue
- Examples
  - stb r5, 0x1000(r3)
  - sth r5, 0x1000(r3)
  - stwx r5, r3, r4

Load/Store Instructions

- Store instructions
  - Three data sizes – byte, half-word, word
  - Two addressing modes – displacement or register index
  - No extension issue
- Examples
  - stb r5, 0x1000(r3)
  - sth r5, 0x1000(r3)
  - stwx r5, r3, r4

Load/Store Instructions

- Store instructions
  - Three data sizes – byte, half-word, word
  - Two addressing modes – displacement or register index
  - No extension issue
- Examples
  - stb r5, 0x1000(r3)
  - sth r5, 0x1000(r3)
  - stwx r5, r3, r4

Load/Store Instructions

- Store instructions
  - Three data sizes – byte, half-word, word
  - Two addressing modes – displacement or register index
  - No extension issue
- Examples
  - stb r5, 0x1000(r3)
  - sth r5, 0x1000(r3)
  - stwx r5, r3, r4

Load/Store Instructions

- Store instructions
  - Three data sizes – byte, half-word, word
  - Two addressing modes – displacement or register index
  - No extension issue
- Examples
  - stb r5, 0x1000(r3)
  - sth r5, 0x1000(r3)
  - stwx r5, r3, r4
**In Class Exercise**

- Convert the following code to assembly using load and store instructions
  
  ```
  char myValue; // located at memory location 0x2000 0000
  char *myAddr;
  myAddr = (char *) 0x1000 0000;
  myVal = *(pMemory);
  ```

  **Answer**
  
  ```
  lis r3, 0x1000
  lis r4, 0x2000
  lbz r5, 0(r3)
  stb r5, 0(r4)
  ```

**Arithmetic in Assembly**

- PowerPC: register-register architecture
  - Arithmetic, logic, and other operations only performed on register and immediate operands
  - Memory operands must be loaded into registers for operations
  - Alternative: register-memory architecture (Intel processors CISC)

  Almost all arithmetic instructions use two sources and one destination
  - `opcode rD, rA, rB ; rD = rA op rB`
  - `opcode rD, rA, IMM ; rD = rA op IMM`

**Arithmetic and Bitwise Operations in Assembly**

- Common arithmetic operations: add, subf, mul, div
- Common bitwise logic: and, or, xor, nand

  **Examples:**
  - `add r5, r3, r4 ; r5 = r3 + r4`
  - `addi r5, r3, 0x100 ; r5 = r3 + 0x100`
  - `subf r5, r3, r4 ; r5 = r4 - r3`
  - `or r5, r3, r4 ; r5 = r3 | r4`

**Shifting in Assembly**

- Logic and arithmetic shifts
  - `slw: shift left word`
  - `srw: shift right word`
  - `sraw: shift right algebraic word (arithmetic shift right)`

  **Examples**
  - `slw r5, r3, r4 ; shift left, r4 gives the # of bits`
  - `srw r5, r3, 1 ; shift right, fill sign bit at the left`
  - `sraw r5, r3, 1 ; shift left by one bit`
  - `rlwinm rA, rS, SH, MB, ME - Rotate Left Word Immediate then AND with Mask`

  **Examples:**
  - Move bits 24-31 in rS to bits 0-7 in rA and set all other bits to zero
    - `SH = 24, MB = 0, ME = 7`
  - Clear the low order 8 bits of a register
    - `SH = 0, MB = 0, ME = 24`

**Assembly Example**

<table>
<thead>
<tr>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>lwz r3, 4(r13) ; load x1</td>
</tr>
<tr>
<td>lwz r4, 8(r13) ; load x2</td>
</tr>
<tr>
<td>add r5, r3, r4 ; x1 + x2</td>
</tr>
<tr>
<td>lwz r3, 12(r13) ; load y1</td>
</tr>
<tr>
<td>lwz r4, 16(r13) ; load y2</td>
</tr>
<tr>
<td>add r6, r3, r4 ; y1 + y2</td>
</tr>
<tr>
<td>subf r5, r3, r6 ; minus</td>
</tr>
<tr>
<td>addi r3, r3, 100 ; ; add 100</td>
</tr>
<tr>
<td>stw r3, 20(r13) ; store sum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every instruction is encoded into 32-bit binary</td>
</tr>
</tbody>
</table>
| `opcode D D A A IMMimm`

<table>
<thead>
<tr>
<th>Opcode</th>
<th>6-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>op rD, rA, IMM</code> (arithmetic/logic with one immediate)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>op rD, d(rA)</code> (load/store using displacement)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>6-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>11-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>op rD, rA, rB</code> (arithmetic/logic using three registers)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>op rD, rA, rB</code> (load with register indexed)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>op rS, rA, rB</code> (store with register indexed)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Instruction Format**

- What is stored for "addi r3, r4, 0x64"? (opcode for addi 0x0E)
  - Binary: 001110 00011 00100 000000001100100
  - Hex: 0x3864
- What is stored for "lwz r3, r4, r5" (Opcode = 0x1F)
  - Binary: 011111 00011 00100 00101 0000010111 0
  - 0x7C64

**XER Overflow Bits**

- Use and update XER[SO] and XER[OV]
  - addx – italic x indicates additional features available
  - addo, subo – affect XER (SO and OV)
- Example:
  - addo r5, r3, r4
- Instruction Format:

<table>
<thead>
<tr>
<th>OEX</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>0x10A</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0010</td>
<td>OEX</td>
</tr>
</tbody>
</table>

- OE=1 for addo – XER[SO] and XER[OV] are affected
- Note: Italic x can also be ".", "o", or "o."
  - "." indicates we want to use the condition register (RC = 1)
  - We will revisit this when we discuss program flow control and branching

**XER Carry**

- Instructions to use and update carry bit XER[CA]
  - addc – “add carrying”, update carry bit
  - addcx – “add extended”, uses carry bit
  - again the x could be null, ".", "o", or "o."
- Example:
  - long long x, y, z;
  - z = x + y;
  - lwz r3, 0(r13); ; load r3 with upper word of x
  - lwz r4, 4(r13); ; load r4 with lower word of x
  - lwz r5, 8(r13); ; r5 = y@h
  - lwz r6, 12(r13); ; r6 = y@l
  - addc r7, r4, r6 ; add lower words x@l + y@l; if carryout, set XER[CA]=1
  - addc r8, r3, r5 ; r8 = r3+r5+XER[CA]
  - How should we store the result (r7 and r8) to memory?

**Accessing Arrays in Assembly**

- Use register indexing to access arrays in assembly
  - Load register with starting address
  - Use proper offset to access array elements

<table>
<thead>
<tr>
<th>C code Example</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>char charArr[20]; begin at 0x2000 0000</td>
<td>li r30, 0x2000 ; r30 &lt;- 0x2000 0000</td>
</tr>
<tr>
<td>int intArr[20]; begin at 0x2000 1000</td>
<td>li r31, 0x2000 ; r31 &lt;- 0x2000 1000</td>
</tr>
<tr>
<td>charArr[0] = 10;</td>
<td>addi r31, r31, 0x10</td>
</tr>
<tr>
<td>charArr[19] = 20;</td>
<td>li r31, 0x20</td>
</tr>
<tr>
<td>intArr[0] = 100;</td>
<td>li r31, 0x100</td>
</tr>
<tr>
<td>intArr[19] = 200;</td>
<td>li r31, 0x200</td>
</tr>
</tbody>
</table>

**Using Pointers in Assembly**

- Similar to using arrays
  - Load pointer address into register
  - Use offset or updated address to access elements

<table>
<thead>
<tr>
<th>C code Example</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *myAddr = (int * )0x2000 0000;</td>
<td>lis r31, 0x2000 ; r31 &lt;- 0x2000 0000</td>
</tr>
<tr>
<td>*myAddr = 10;</td>
<td>li r0, 10 ; r0 &lt;- 10</td>
</tr>
<tr>
<td>*(myAddr + 10) = 20;</td>
<td>add r0, r0, 10 ; r0 &lt;- r0 + 10</td>
</tr>
<tr>
<td>*(myAddr)++;</td>
<td>stw r0, 0(r31) ; *(myAddr)++;</td>
</tr>
<tr>
<td>*(myAddr) = 30;</td>
<td>li r0, 30 ; r0 &lt;- 30</td>
</tr>
<tr>
<td>*(myAddr + 20) = 40;</td>
<td>add r0, r0, 20 ; r0 &lt;- r0 + 20</td>
</tr>
<tr>
<td>*(myAddr + 40);</td>
<td>stw r0, 40(r31) ; *(myAddr + 40) = 40;</td>
</tr>
<tr>
<td>*(myAddr++) = 50;</td>
<td>add r0, r0, 40 ; r0 &lt;- r0 + 40</td>
</tr>
<tr>
<td>*(myAddr) = 60;</td>
<td>stw r0, 60(r31) ; *(myAddr) = 60;</td>
</tr>
</tbody>
</table>
Writing an Assembly Program

- Using Compiler directives
- Including other files
  - `include "filename.h"` – includes the file specified by filename
  - `export label` – Allows you to call the particular assembly code section from another file – place at top of asm code
- Function “function name” startLabel, length – specifies that the subroutine “function name” begins at startLabel and is length bytes long (for debug purposes – very helpful)
- Assembly body
  - `.text` – specifies the executable code section
  - `.data` – specifies a read-write data section
- More directives in the Code Warrior Assembler Guide on the links page of the website

Program Execution

- Now we can do load/store operations and arithmetic operations
- How does the processor know what instruction to do next?
- How do we deal with conditional statements and loops?
- What about function calls?
- First need to know how the processor executes instructions

Basic Architecture Example

- Calculator example
  - Fetch/Decode – user enters the operands and operation (instruction) and the processor determines what operation we want to do
  - Read – operands must be accessed from storage (registers)
  - Execute – processor executes the desired operation
  - Write result – result is written to storage (screen)
- Many different architectures to facilitate this – you can learn about this in Cpre 305 and 483

Machine-level Execution

- Program operation
  - “Fetch” a 32-bit instruction from memory at PC (program counter) address
  - “Decode” the instruction – what operation are we going to perform
  - “Read” operands – registers or immediate values
  - “Execute” Perform data operation or address calculation
  - “Write” register/read memory/store memory, and update PC

Asm Code Using Labels and Directives

```
.export StartAsm ;export the StartAsm label so other files can see this function
.function "StartAsm", PPC_Start_Asm, PPC_End_Asm
.text ; begin the executable section
PPC_Start_Asm: ;PPC_Start_Asm and StartAsm have the same address
StartAsm:
  lis r3, DataSeg@h ; r3 = Upper Address
  ori r3, r3, DataSeg@l ; r3 = r3 | lower 16 bits of address
  lwz r4,0(r3) ; load the contents of address r3+0 into r4
  blr ; Return back to calling function
PPC_End_Asm: ; memory allocation section
.data
  DataSeg:
    .word 0x40
```

Changing Program Sequence in C

- If statement
  ```
  if (n > 0) {
    ...
  } else {
    ...
  }
  ```
- While loop
  ```
  while (s != NULL) {
    ...
  }
  ```
- For loop
  ```
  for(i=0; i<N; i++){
    ...
  }
  ```
Changing Program Sequence in ASM

- Branches and Jumps – Change the program control to a given address
- Conditional branch
  - A branch instruction that comes with a condition
  - If the condition is true, the branch is taken; otherwise, the branch is not taken
- Unconditional branch (jump)
  - The branch is always taken
- Branch Target address – The address of the next instruction if the branch is taken
- Example: beq target – next PC gets (1) target address, if “EQ” is true; (2) PC+4, otherwise

Control Instructions

C Program
if (x < y)
  z = 1;
else
  z = 0;

Assembly
```
 cmpw r3, r4
 bge Skip
 li r31, 1
 b end
Skip: li r31, 0
end: ...
```

- Assume that r3 ← x, r4 ← y and r31 ← z
- bge – branch if greater than or equal
- li r31, 1 is a simplified mnemonic for addi r31, r0, 1

Condition Register

- 32 bit register that reflects the result of certain operations and aids in testing and branching
- Grouped into eight 4-bit fields
- Four condition bits about the result of arithmetic/logic instructions
  - LT: Less than zero? (negative)
  - GT: Greater than zero? (positive)
  - EQ: Equal zero?
  - SO: Overflow occurred? (Summary of Overflow – copy of XER[SO])

Comparison Instructions

- Use comparison instructions to set condition register
- Compare signed words
  - cmpw rA, rB ; set CR0 for signed comparison of rA and rB
- CR0 fields
  - LT = 1 if rA < rB
  - GT = 1 if rA > rB
  - EQ = 1 if rA = rB
  - SO – summary of overflow

Comparison Instructions

- Compare unsigned words (compare logical)
  - cmpwl rA, rB ; set CR0 for unsigned comparison
- CR0 fields are the same
- Comparisons can be done with immediate values
  - cmpwi r3, 200 ; set CR0 as for signed r3-200
  - cmplwi r3, 300 ; set CR0 as for unsigned r3-300
- Comparison and conditional branch can use any CR field
  - cmp cr1, r3, r4 ; set condition bits of CR1
  - blt cr1, target ; taken if CR1.LT = 1
Branch Instructions

- Use branch instructions to determine what action to take once the comparison is done.
- Branch instructions use the CR fields to make decision.
- Examples:
  - blt target; branch taken if LT = 1 (less than)
  - bge target; branch taken if GT = 1 (greater than)
  - bne target; branch taken if EQ = 0 (not equal)
  - bge target; branch taken if LT = 0 (greater than or equal)
  - ble target; branch taken if GT = 0 (less than or equal)

Example Revisited

```
C Program
if (x < y)
    z = 1;
else
    z = 0;
```

```
Assembly
CMPW r3, r4
BGE Skip
LI r31, 1
B END
Skip: LI r31, 0
END:
```

Assume that r3 ← x, r4 ← y and r31 ← z.
- bge = branch if greater than or equal.
- li r31, 1 is a simplified mnemonic for addi r31, r0, 1.

Branch Example

- Specifying the target address
- Branch based upon displacement
- Branch using labels

C program example
```
int x, y, z;
if(x < y)
    z = 1;
else
    z = 0;
```

Codewarrior Example

- Assembly code 1
  ```
  ; load address of dataseg into r3
  LI r3, DataSeg@h
  ORI r3, r3, DataSeg@l
  Lwz r30, 0(r3) ; r30 ← 0x40
  Lwz r31, 4(r3) ; r31 ← 0x50
  CMPW r30, r31 ; compare r30 and r31
  BLT $+12 ; branch PC+12 if r30<r31
  LI r4, 1  ; load 1 into r29
  B $+8    ; branch to PC + 8
  LI r4, 0  ; location of first branch
  BLR ; branch to link register
  DataSeg:
  .word 0x40
  .word 0x50
  ```

- Assembly code 2
  ```
  ; load address of dataseg into r3
  LI r3, DataSeg@h
  ORI r3, r3, DataSeg@l
  Lwz r31, 0(r3) ; r30 ← 0x40
  Lwz r30, 4(r3) ; r31 ← 0x50
  CMPW r30, r31 ; compare r30 and r31
  BLT SkipElse ; branch to SkipElse if r30<r31
  LI r4, 1  ; load 1 into r29
  B End    ; branch to label End SkipElse:
  LI r4, 0  ; location of first branch
  BLR ; branch to link register
  DataSeg:
  .word 0x40
  .word 0x50
  ```
**QC2 – In Class Exercise**

- Write the assembly code for the following C program – you do not need to create space in memory for the variables
  - **Steps**
    1. Choose registers for your variables (sum and i)
    2. Assume r1 has the base address of X[]
    3. Do any initializations that are required
    4. Loop coding has multiple solutions – suggestion is to branch to a comparison point, do comparison and if necessary branch back to loop body
    5. Do loop body, increment the counter and do comparison again
    6. Branch to loop body again if necessary

```c
int sum = 0;
int X[100];
int i;
for (i = 0; i < 100; i++)
    sum += X[i];
```

**Looping in Assembly**

- **C code**
  ```c
  int sum = 0;
  int X[100];
  int i;
  for (i = 0; i < 100; i++)
      sum += X[i];
  ```

- **Assembly**
  ```asm
  li    r30, 0 ; sum=0 ; r30 <- sum
  li    r31, 0 ; i=0; r31 <- i
  b       cmp;
  loop: slwi   r0, r31, 2 ; r0=i*4 – array offset
          addi   r3, SP, 8 ; r3 <- X[0] address 8(SP)
          lwzx  r0, r3, r0 ; r0 <- X[i] – X[0]+offset
          add   r30, r30, r0 ; sum+=X[i]
          addi  r31, r31, 1 ; i++
  cmp:   cmpwi  r31, 0x0064 ; 0x64 = 100
          blt loop
  ```
  (generated by CodeWarrior and then revised)

- Which part of this code costs the most to execute?
- Can we optimize the assembly code?

**Better Loop Programming**

- Optimizing the loop body – remove 2 instructions
  - Base address calculation for X[i] is moved out of the loop body
  - Loop counter (i) is incremented by 4 instead of 1 – acts as both the loop count and as array offset

```asm
li       r30, 0 ; sum=0 ; sum
li       r31, 0 ; i=0; r31<-i
addi   r3, SP, 8 ; r3 <- X[0] address
b       cmp
loop:    lwzx r0, r3, r31 ; load X[i]
          add   r30, r30, r0 ; sum+=X[i]
          addi  r31, r31, 4 ; increase i
  cmp:   cmpwi  r31, 0x0190 ; 0x190=400
          blt loop
```

**Better Yet**

- Remove the compare statement

```asm
li       r30, 0 ; sum=0 ; sum
li       r31, 0x18C ; i=396; r31<-i ; i.e X[99];
addi   r3, SP, 8 ; r3 <- X[0] address
b       cmp
loop:    lwzx r0, r3, r31 ; load X[i]
          add   r30, r30, r0 ; sum+=X[i]
          addi  r31, r31, -4 ; decrement i
cmp:   bge  loop ; if i >= 0 branch
```

**Stack Example**

- **SP points to the top of the stack**
- **Last used memory location**
- **SP = 0x204F FFF8**
- Stack grows negatively with respect to memory addresses

- **Stack Example**
  ```asm
  .section .data
  32-bit width
  0x204F FFFF
  0x204F FFF4 -4(SP)
  Mem Grow
  Lower Memory Addresses
  ```
  ```asm
  .section .text
  Stack Grows
  ```
  ```asm
  .section .data
  32-bit width
  0x204F FFFF
  0x204F FFF4 -4(SP)
  Mem Grow
  Lower Memory Addresses
  ```
  ```asm
  .section .text
  Stack Grows
  ```
**Stack Example**

- Push double word
  - 0x20000000 0x10000000
- 4 Step process – assume r3, r4
  - Add SP, SP, -4
  - sub r4, r0, (SP)
  - Add SP, SP, -4
  - sub (r3, 30(SP))
- Pushed LSW first – Why?
  - Big-endian convention indicates MSB is at lowest address
  - Stack grows toward lower address, LSW gets pushed first
- SP is now at address 0x204F FFFC

**Stack Example**

- To pop a value
  - Read current SP location
  - Update SP to previous location
- 4 step process for our double word
  - lwz r3, 0(SP)
  - Addi SP, SP, 4
  - lwz r4, 0(SP)
  - Addi SP, SP, 4
- After reading our double word, SP is pointing to address 0x204F FFF8

**Stack Notes**

- Pushing and Popping
  - Should be symmetric: what goes on, must come off
  - Popping an item from the stack does not clear the memory location
  - What are local variables declared in C initialized to?
  - For PPC, SP is incremented at least +/- 4 bytes at a time
    - SP must be word aligned where word boundaries are evenly divisible by 4
    - However it is possible to access individual bytes using address offset i.e. 1(SP), 11(SP)

**EABI Rules**

- What should be put on the stack when entering a function?
- Main question is how to keep things consistent
- Consider you want to write a function in assembly to be called from your C code
  - How are values passed to the function?
  - What about return values?
  - What registers should be saved?
  - How do we return to the calling function?
- The compiler follows rules that you are also expected to follow
- Look at EABI rules and discuss “stack frame”

**EABI Register Rules**

- Volatile registers need not be preserved by called functions
- Nonvolatile registers must be returned to the caller as they were received – save nonvolatile register values in the prologue code and restore them in the epilogue code

<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 - R2</td>
<td>Nonvolatile</td>
<td>Stack boundaries</td>
</tr>
<tr>
<td>R3 - R11</td>
<td>Nonvolatile</td>
<td>Load/store data from memory</td>
</tr>
<tr>
<td>R12-S7</td>
<td>Nonvolatile</td>
<td>Global variables, stack frame</td>
</tr>
<tr>
<td>R8 - R11</td>
<td>Nonvolatile</td>
<td>Local variables, stack frame</td>
</tr>
<tr>
<td>R0 - R7</td>
<td>Volatile</td>
<td>Load/store data from memory</td>
</tr>
<tr>
<td>R12 - R15</td>
<td>Nonvolatile</td>
<td>Controls</td>
</tr>
<tr>
<td>R12-S7</td>
<td>Nonvolatile</td>
<td>Stack frame</td>
</tr>
<tr>
<td>R8 - R11</td>
<td>Volatile</td>
<td>Local variables, stack frame</td>
</tr>
<tr>
<td>R0 - R7</td>
<td>Volatile</td>
<td>Load/store data from memory</td>
</tr>
<tr>
<td>R12 - R15</td>
<td>Volatile</td>
<td>Controls</td>
</tr>
<tr>
<td>CCR and CCRX</td>
<td>Nonvolatile</td>
<td>Status flags</td>
</tr>
<tr>
<td>Other x86 registers</td>
<td>Nonvolatile</td>
<td>Local information</td>
</tr>
</tbody>
</table>

**Register Usage**

- Choosing between volatile and non volatile registers
  - Programmers choice
  - Good practice is to choose non volatile registers for important local information
  - Volatile registers should only be used for parameter passing and return values
- What happens if you are interrupted during program execution?
Stack Frame (SF)
- Organizes or delineates a function’s stack space
- Any function that either calls another function or modifies a nonvolatile register must create a SF
- EABI defines conventions for SF creation and usage
  - Parameter passing
  - Nonvolatile register preservation
  - Local variable storage
  - Function return – linkage
- SF is created by placing the various data onto the stack in a consistent manner
- If a function is a leaf function (meaning it calls no other functions) and does not modify any nonvolatile registers an SF is not needed

Creating a Stack Frame
- Prologue code creates a new stack frame upon entry to a function
  - New frame created adjacent to the most recently allocated frame
  - SP is decremented one time by the total amount of space required by the function (for local variables, non-volatile registers and a few others)
  - Use store with update (stwu) to insure the SP update is not interrupted
- Epilogue code destroys the stack frame before exiting
  - Sets return register value (if function returns anything)
  - Restores nonvolatile registers
  - De-allocates current stack frame by incrementing SP - Memory is not cleared
  - Restores the link register (if necessary)
  - Returns to the calling function

Memory View of Stack Frames
- 2 level deep function calling example
  - Time 1 – Function A exists and calls function B
  - Time 2 – B’s prologue code has created B’s stack frame
  - Time 3 – B has called C and C’s prologue code has executed
  - Time 4 – C has terminated and C’s epilogue code has destroyed its frame by incrementing the SP

Stack Frame Creation
- Up to 8 parameters can be passed in r3-r10
- Return result is passed back in r3
- I have shown 1 solution – lets see how CodeWarrior handles this

Creating a Stack Frame
- Prologue code creates a new stack frame upon entry to a function
  - New frame created adjacent to the most recently allocated frame
  - SP is decremented one time by the total amount of space required by the function (for local variables, non-volatile registers and a few others)
  - Use store with update (stwu) to insure the SP update is not interrupted
- Epilogue code destroys the stack frame before exiting
  - Sets return register value (if function returns anything)
  - Restores nonvolatile registers
  - De-allocates current stack frame by incrementing SP - Memory is not cleared
  - Restores the link register (if necessary)
  - Returns to the calling function

Stack Frame Details
- Programmers view of the stack frame
  - Every stack frame must be at least 8 bytes (and a multiple of 8 bytes)
  - LR Save Word – place to store the link register of calling program
  - Back Chain Word – place to store the old SP value (allows programmer to access callers stack – not frequently used)

Stack Frame Diagrams
- My stack diagram
- CodeWarrior stack
  - Very similar Stack Frames just different interpretations