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

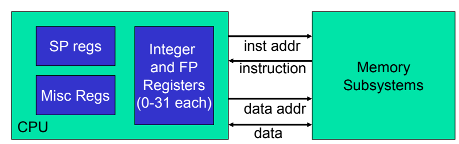
Assembly Language Programming

- 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
  - Brach 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



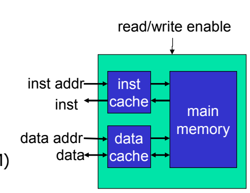
- Registers
  - Most frequently used data
  - Vary fast access – operates at processor clock speed

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



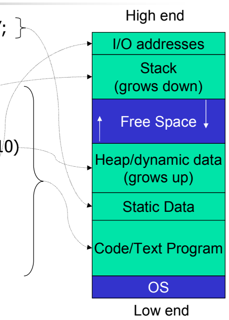
Memory Address Space

- 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

IOWA STATE UNIVERSITY

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);
}
```



IOWA STATE UNIVERSITY

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

IOWA STATE UNIVERSITY

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

IOWA STATE UNIVERSITY

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

IOWA STATE UNIVERSITY

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

IOWA STATE UNIVERSITY

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(r31) # r5 <- value at $r31 + 0x1000
addi r5, r5, 100     # add 100 (0x64)
stw r5, 0x1000(r31) # store back
```

IOWA STATE UNIVERSITY

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

IOWA STATE UNIVERSITY

Load/Store Instructions

- Assume $r3 = 0x2000 0000, mem(0x20000200) = 0x1234 5678
- Loading registers from memory – 3 sizes
  - Load byte
    - lbz r5, 0x200(r3) # r5 = 0x0000 0012
    - lbz 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

IOWA STATE UNIVERSITY

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 ffff
  - lha r5, 0x1000(r3) ; r5 = 0xffff ffff
  - z – zero, a – algebraic
- What load instruction to use for m and n?
  - short m;
  - unsigned short n;

IOWA STATE UNIVERSITY

Load/Store Instructions

- Suppose r3 = 0x2000 0000,
  r4 = 0x0000 1000
  mem(0x2000 1000) = 0x8765 4321
- What will be the value in register r5?
  - lbz r5, 0x1000(r3)
  - lba r5, 0x1000(r3)
  - lhzx r5, r3, r4
  - lhax r5, r3, r4
- Also have load with update option which updates the base register
  - Lhzu r5, 0x1000(r3) ; r5 = 0x0000 4321, r3 = 0x2000 1000
- Pseudo Instructions – load immediate shifted
  - lis r3, 0x2000 -> addis r3, r0, 0x2000

IOWA STATE UNIVERSITY

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

IOWA STATE UNIVERSITY

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
  - sraw r5, r3, r4 ; shift right, fill sign bit at the left
  - slwi r5, r3, 1 ; shift left by one bit
- rlwinm rA, rS, SH, MB, ME - Rotate Left Word Immediate then AND with Mask
  - rA <- rS rotated by SH bits ANDed with the Mask
  - Mask value is 1's from MB to ME and 0s elsewhere
- 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

<u>Basic operations</u>
```
int sum;
int x1, x2;
int y1, y2;
…
sum =  (x1+x2)-(y1+y2)+100;
```

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

Instruction Format

- Every instruction is encoded into 32-bit binary

opcode	D	A	IMM/d
6-bit	5-bit	5-bit	16-bit

op rD, rA, IMM (arithmetic/logic with one immediate)
op rD, d(rA) (load/store using displacement)

opcode	D/S	A	B	Other bits
6-bit	5-bit	5-bit	5-bit	11-bit

op rD, rA, rB (arithmetic/logic using three registers)
op rD, rA, rB (load with register indexed)
op rS, rA, rB (store with register indexed)

Instruction Format

- What is stored for "addi r3, r4, 0x64"? (opcode for addi 0x0E)
  - Binary: 001110 00011 00100 0000000001100100
  - Hex: 0x3864 0064
- What is stored for "lwz r3, r4, r5 (Opcode = 0x1F)
  - Binary: 011111 00011 00100 00101 0000010111 0
  - 0x7C64 282E

IOWA STATE UNIVERSITY

Integer Exception Register (XER)

- Sometimes integer arithmetic operations can cause problems
- What happens if the result is too large?
- What about operations on long integers (declared long long in gcc)?
- Use the information in the XER register
  - XER[SO] – summary overflow indicates overflow and remains set until explicitly cleared
  - XER[OV] – set if overflow occurs on current instruction
  - XER[CA] – indicates a carry out occurred
  - Byte count is the number of bytes to be transferred for lswx – load string word indexed and stswx

0	1	2		25-31
SO	OV	CA	00…0	Byte count

IOWA STATE UNIVERSITY

XER Overflow Bits

- Use and update XER[SO] and XER[OV]
  - add*x* – the italic *x* indicates additional features available
  - add*o*, subf*o* – affect XER (SO and OV)
- Example

  addo r5, r3, r4
- Instruction Format:

0x1F	D	A	B	OE	0x10A	RC

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

IOWA STATE UNIVERSITY

XER Carry

- Instructions to use and update carry bit (XER[CA])
  - addc*x* – "add carrying", update carry bit
  - adde*x* – "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
adde r8, r3, r5   ; r8 = r3+r5+XER[CA]
```

- How should we store the result (r7 and r8) to memory?

IOWA STATE UNIVERSITY

Accessing Arrays in Assembly

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

C code Example

```
char charArray[20]; begins at 0x2000 0000
int intArray[20]; begin at 0x2000 1000

charArray[0] = 10;
charArray[19] = 20;
intArray[0] = 100;
intArray[19] = 200;
```

Assembly Code

```
lis r30, 0x2000          ; r30 <- 0x2000 0000
lis r31, 0x2000          ; r31 <- 0x2000 0000
addi r31, r31, 0x1000    ; r31 <- 0x2000 1000
li r0, 10                ; r0 <- 10
stb r0, 0(r30)           ; charArray[0] <- 10
li r0, 20                ; r0<- 20
stb r0, 19(r30)          ; charArray[19] <- 20
li r0, 100               ; r0 <- 100
stw r0, 0(r31);          ;intArray[0] <- 100
li r0, 200               ;r0 <- 200
stw r0, 76(r31);         ;intArray[19] <- 200
```

IOWA STATE UNIVERSITY

Using Pointers in Assembly

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

C code Example

```
int *myAddr = (int * )0x2000 0000;

*myAddr = 10;
*(myAddr + 10) = 20;
myAddr++;
*myAddr = 30;
```

Assembly Code

```
lis r31, 0x2000          ; r31 <- 0x2000 0000

li r0, 10                ; r0 <- 10
stw r0, 0(r31)           ; *myAddr <- 10
li r0, 20                ; r0 <- 20
stw r0, 40(r31)          ; *myAddr <- 20
addi r31, r31, 4         ; myAddr++
li r0, 30                ; r0 <- 30
stw r0, 0(r31)           ; *myAddr <- 30
```

IOWA STATE UNIVERSITY

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

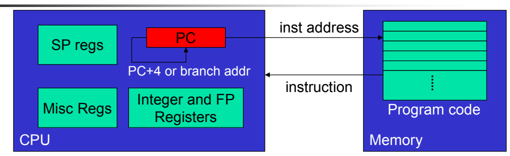
Asm Code Using Labels and Directives

```
.export StartAsm     ;export the StartAsm label so other files can see this function
.function "StartAsm", PPC_Start_Asm, PPC_End_Asm-PPC_Start_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:

.data                ; memory allocation section
DataSeg:
 .word 0x40
```

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

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

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

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	Assembly

<u>C Program</u>
if (x < y)
z = 1;
else
z = 0;
	<u>Assembly</u>
	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])

CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7

eight fields, 32-bit

LT	GT	EQ	SO

4-bit

Condition Register

- Condition register can be used for arithmetic and logic instructions
  - Instructions with a "." will modify CR0
  - Not available for all instructions – check reference manual
- Can make branch decisions based upon arithmetic operation
- Example

  subf. r5, r3, r4 ; r5 ← r4 – r3

  beq target ;branch to target if cr0[eq] is set i.e r3=r4

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)
  - cmplw 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
  - cmpw 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)
  - bgt *target* ; taken if GT = 1 (greater than)
  - beq *target* ; taken if EQ = 1 (equal)
  - bne *target* ; taken if EQ = 0 (not equal)
  - bge *target* ; taken if LT = 0 (greater than or equal)
  - ble *target* ; 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
lis  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
lis  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
End:
 blr ; branch to link register

DataSeg:
 .word 0x40
 .word 0x50
```


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
lis  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
lis  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
End:
 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
    - Choose registers for your variables (sum and i)
      - Assume r1 has the base address of X[]
    - Do any initializations that are required
    - Loop coding has multiple solutions – suggestion is to branch to a comparison point, do comparison and if necessary branch back to loop body
    - Do loop body, increment the counter and do comparison again
    - Branch to loop body again if necessary

```
int sum = 0;
int X[100];
int i;

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

Looping in Assembly

C code

```
int sum = 0;
int X[100];
int i;

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

Assembly

```
        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

Assembly

```
        li     r30, 0            ; sum=0  ; sum⇔r31
        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
```

- Optimizing the loop body – remove 2 instructions
  - Base address calculation for X[] 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

Better Yet

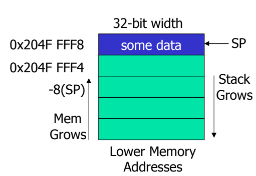
Assembly

```
        li     r30, 0            ; sum=0  ; sum⇔r31
        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
```

- Remove the compare statement

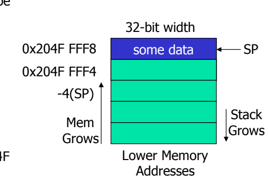
Stack Example

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



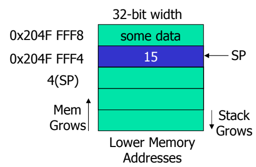
Stack Example

- Push the value "15" to the stack
  - How much space to use?
  - Most cases the minimum space used (by compilers) is the width of the memory regardless of type
  - Exception is when pushing data structures – i.e. char array only uses 1 byte for each element
- Address 0x204F FFF8 gets the value 15
- PPC – two step process
- Assume r0 has value 15
  - addi SP, SP -4
  - stw r0 0(SP)
  - SP now points to address 0x204F FFF4



Stack Example

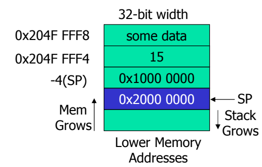
- Push double word 0x20000000 10000000
- 4 Step process – assume r3, r4 has 0x20000000, 0x10000000
  - addi SP, SP, -4
  - stw r4, 0(SP)
  - addi SP, SP, -4
  - stw r3, 0(SP)
- Pushed LSW first – Why?
  - Big-endian convention indicates MSB is at lowest address
  - Stack grows toward lower address, LSW gets pushed first
- SP is Now at address 0x204F FFEC



IOWA STATE UNIVERSITY

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 FFF4



IOWA STATE UNIVERSITY

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)

IOWA STATE UNIVERSITY

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"

IOWA STATE UNIVERSITY

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

Register	Type	Used for:
R0	Volatile	Language Specific
R1	Dedicated	Stack Pointer (SP)
R2	Dedicated	Read-only small data area anchor
R3 - R4	Volatile	Parameter passing / return values
R5 - R10	Volatile	Parameter passing
R11 - R12	Volatile	
R13	Dedicated	Read-write small data area anchor
R14 - R31	Nonvolatile	
F0	Volatile	Language specific
F1	Volatile	Parameter passing / return values
F2 - F8	Volatile	Parameter passing
F9 - F13	Volatile	
F14 - F31	Nonvolatile	
Fields CR2 - CR4	Nonvolatile	
Other CR fields	Volatile	
Other registers	Volatile	

IOWA STATE UNIVERSITY

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?

IOWA STATE UNIVERSITY

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

IOWA STATE UNIVERSITY

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

IOWA STATE UNIVERSITY

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

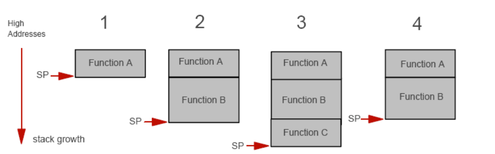


Figure 1 - Stack Frame creation and destruction

IOWA STATE UNIVERSITY

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)

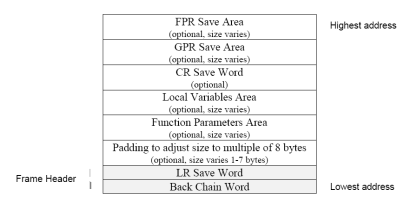


Figure 2 - EABI Stack Frame

IOWA STATE UNIVERSITY

Stack Frame Creation

```
void main()
{
     int var1, var2, result;
     result = function1(var1, var2);
     return;
}
int function1(int var1, int var2)
{
     int temp1;
     temp1 = var1 + var2;
     return temp1;
}
```

```
stwu  SP,-24(SP) ;create SF for main
lwz    r3,8(SP)    ;load param1, r3<-var1
lwz    r4,12(SP)  ;load param2, r4<-var2
bl      function1  ;call function1
stw    r3,16(SP)  ;store ret value, result<-r3

function1:
stwu  SP,-16(SP); create SF for function1
add   r3,r3,r4     ;r3<-var1 + var2
stw   r3,12(SP)   ;store temp1 into SF
addi  SP,SP,16    ;restore SP
blr                      ;branch to LR (main)
```

- 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

IOWA STATE UNIVERSITY

Stack Frame Diagrams

- My stack diagram

	LR (main)	
SP before function1	BCW (main)	
-4(SP)	temp1 (local var)	12(SP)
-8(SP)	Padding	8(SP)
-12(SP)	LR (function1)	4(SP)
-16(SP)	BCW (function1)	SP after function1

- CodeWarrior stack

	LR (main)	
SP before function1	BCW (main)	
-4(SP)	GPR31 (save)	12(SP)
-8(SP)	Padding	8(SP)
-12(SP)	LR (function1)	4(SP)
-16(SP)	BCW (function1)	SP after function1

- Very similar Stack Frames just different interpretations

IOWA STATE UNIVERSITY