Introduction

Z. Jerry Shi
Department of Computer Science and Engineering
University of Connecticut

CSE4903: Microprocessor Laboratory
What I think you should already know

• How to use computers
  – Windows (and Linux)

• One type of assembly languages
  • X86, MIPS, PA-RISC, etc.

• C language

• Digital Logic design (CSE2300, i.e., CSE210)
  – Binary numbers, hex numbers, etc.
  – Logical operations
  – Implementation of adder, multiplier, etc.

• Computer architecture (CSE3336)
  – Datapath, pipeline, cache, etc.
What you will learn in this course

• PowerPC architecture and MPC875
• Experiment with architecture features and new workload
• Learn on your own
  – Read and think
  – Trial and error
  • Programming takes time
  • Be patient and persistent
• Solve problems
  – Any instructions to follow?
  – Find your own solutions
  – Any better solutions?
• Teamwork
  – Find a good partner and be a good partner
Learn more and get a better grade

• Final grades
  – 70% : projects
  – 15% for exams
  – 15%: overall performance
• Best students get A
  – Most students get B+, B, or B –
    • Unless you do not work hard
• Prepare before lectures and labs
• Write good reports
  – Follow the lab report guidelines
MPC875 and PowerPC instruction set

Z. Jerry Shi
Department of Computer Science and Engineering
University of Connecticut

CSE4903: Microprocessor Laboratory
MP875 block diagram

Courtesy of Freescale Semiconductor
Reduced instruction set computer (RISC)

- Relatively few instructions
- Instructions have the same length
- Relatively few instruction formats
- Instructions can be executed in one cycle
  - Except for a few instructions like MUL
  - Good for pipeline design
- Relatively few addressing mode
- Operands are stored in registers
  - except for LOAD/STORE instructions
- A large number of registers
## RISC vs. CICS

<table>
<thead>
<tr>
<th></th>
<th>RISC</th>
<th>CISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of instructions</td>
<td>&lt; 100</td>
<td>several hundreds</td>
</tr>
<tr>
<td>Length of instructions</td>
<td>fixed</td>
<td>variable</td>
</tr>
<tr>
<td>Instruction format</td>
<td>few</td>
<td>many</td>
</tr>
<tr>
<td>Execution cycle</td>
<td>normally one cycle</td>
<td>variable</td>
</tr>
<tr>
<td>Addressing mode</td>
<td>&lt; 4</td>
<td>many</td>
</tr>
<tr>
<td>Operands</td>
<td>in registers</td>
<td>can be memory to memory</td>
</tr>
<tr>
<td>Number of registers</td>
<td>many (&gt; 30)</td>
<td>few (2 – 16)</td>
</tr>
<tr>
<td>Hardware implementation</td>
<td>efficient</td>
<td>expensive</td>
</tr>
</tbody>
</table>
Popular processors

- Intel
  - IA32: CISC
  - IA64: RISC
    - Many features borrowed from HP PA-RISC processors
  - XScale: RISC was ARM’s StrongARM2
- AMD
  - Athlon: RISC
    - Using technologies in DEC Alpha 21064
  - Hammer: Athlon with 64-bit extension
- Freescale and IBM
  - PowerPC: RISC (PowerPC, PowerMac, …)
- ARM (Advanced RISC Machines Ltd.)
  - RISC processor core
- SPARC
  - RISC processors designed by Sun, TI, and Fujitsu
Simplified PowerPC core block diagram

Instruction Unit

MMU

Load/Store Unit

LSU

GPR File

ALU
Where are data stored?

- Memory
- **General-purpose registers**
- Special-purpose registers
  - XER, Condition register, Count register, Link register
  - And many more …
- Immediate encoded in instructions
  - 16-bit constants
    - Can be signed or unsigned

How do you move data from one location to another?
Moving data

- Memory to/from GP registers
  - Load and Store
    - lbz, lhz, lwz, … (with u or x)
- Special purpose to/from GP registers
  - Special instructions
    - mfspr and mtspr
    - and many other mf* and mt* (e.g. mftb, mttb ...) instructions
- Immediate to GP registers
  - Use immediate as an operand for arithmetic operations
    - ADDI, ANDI, ORI, …

How about memory to special-purpose register?
Computation

• Normally done with GPR
• Exception: a set of logical operations for Condition registers
  – crand, crandc, cror, …
Registers in a 32-bit PowerPC core

• General-purpose registers
  – 32 registers (R0 to R31)
• Condition registers
  – CR0 to CR7, each 4 bits: LT, GT, EQ, and SO
• XER register
  – Summary overflow(SO), Overflow(OV), and Carry (CA)
• Link register (LR)
• Count register (CTR)
• Other registers
  – Floating-point registers, time base facility, configuration, memory management, exception handling, etc.
General-purpose registers (GPR)

- R0 through R31 are in general-purpose register file
  - R0 is not always 0
  - R0 is 0 in some instructions

```
addi    R1, R0, 10
```

How do you know R0 is treated as 0?

You can use any GPR, but follow calling conventions if your codes will be linked with other codes.
**Condition register (CR)**

- A 32-bit register used for testing and branching
- Grouped into eight 4-bit fields: CR0 – CR7
- Can be accessed with instructions such as crand and cror

---

**Figure 4-1. Condition Register (CR)**

<table>
<thead>
<tr>
<th>CR0</th>
<th>CR1</th>
<th>CR2</th>
<th>CR3</th>
<th>CR4</th>
<th>CR5</th>
<th>CR6</th>
<th>CR7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>19</td>
<td>20</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
<td>27</td>
<td>28</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4-3. Bit Settings for CR0 Field of CR**

<table>
<thead>
<tr>
<th>CR0 Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Negative (LT). Set when the result is negative.</td>
</tr>
<tr>
<td>1</td>
<td>Positive (GT). Set when the result is positive (and not zero).</td>
</tr>
<tr>
<td>2</td>
<td>Zero (EQ). Set when the result is zero.</td>
</tr>
<tr>
<td>3</td>
<td>Summary overflow (SO). This is a copy of the final state of XER[SO] at the completion of the instruction.</td>
</tr>
</tbody>
</table>
PowerPC instruction set architecture

- Integer instructions
- Load and Store instructions
- Flow control instructions
- Trap instructions
- Processor control instructions
- Memory synchronization instructions
- Memory control instructions
- System linkage instructions
Integer instructions

Arithmetic instructions:

\[ \text{add} \quad R1, R2, R3 \quad ; \quad R1 = R2 + R3 \]

Compare instructions:

\[ \text{cmp} \quad cr3,0,R1, R2 \quad ; \quad \text{result will be CR3} \]
\[ \text{cmpw} \quad cr3, R1, R2 \quad ; \quad \text{Mnemonics} \]
\[ \text{cmpw} \quad R1, R2 \quad ; \quad \text{set CR0} \]

Logical instructions:

\[ \text{or} \quad R1, R2, R3 \quad ; \quad R1 = R2 \text{ } \lor \text{ } R3 \]

Rotate and shift instructions:

\[ \text{rlwnm} \quad R1,R2,R3,0,31 \quad ; \quad R1 = R2 \text{ } \lll \text{ } R3 \]
\[ \text{rotlw} \quad R1,R2,R3 \quad ; \quad \text{Mnemonics} \]

Read Appendix F of “programming environments” for more information about mnemonics
Load and store instructions

Three addressing modes for Load and Store
Here are examples for load words, check instruction reference for loading data of other sizes, such as bytes and half words.

Register indirect addressing:

```assembly
lwz R1, (R2) ; R1 = *R2
             ; EA = R2
```

Register indirect with immediate index addressing:

```assembly
lwz R1, 4(R2) ; R1 = *(R2 + 4)
               ; EA = R2 + 4
               ; R0 is always 0
```

Register indirect with index addressing:

```assembly
lwz R1, R2, R3 ; R1 = *(R2 + R3)
                ; EA = R2 + R3
```
Load and store instructions (2)

- Use three addressing modes to load and store
  - byte (8-bit), half word (16-bit), word (32-bit)
- The base register may be updated
  - Efficient to do operations like *p++
Big-endian

\[
R1 = 0x01020304
\]

After R1 is stored into memory (0xAA00):

\[
\begin{align*}
0xAA00: & \quad 01 \\
0xAA01: & \quad 02 \\
0xAA02: & \quad 03 \\
0xAA03: & \quad 04
\end{align*}
\]

After a half word is loaded into R2:

\[
R2 = 0x00000102
\]

Big-endian in registers too!

Bit 0 is the left-most bit (MSB).
Branch and flow control instructions

• Branch instructions
  – \( bx : b, ba, bl, bla \)
    • Absolute address or relative address?
    • Save the next instruction address in LR?

• Branch conditional instructions
  – \( bcx : bc, bca, bcl, bcla \)

• Branch conditional to link register
  – bclr, can be used as return

• Branch conditional to count register
Control flow

• Branch
  – b, bc, …

• Loop
  – Use conditional branch instructions
    • Count register
    • GPR (compare and test)

• Function call and return
  – bl saves the return address in Link register
  – blr goes to the address stored in Link register
    • blr == bclr 20, 0 (green book, PE:pages F-8 and F-10).

• Exceptions
Example setting CR0

Table 4-3. Bit Settings for CR0 Field of CR

<table>
<thead>
<tr>
<th>CR0 Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Negative (LT). Set when the result is negative.</td>
</tr>
<tr>
<td>1</td>
<td>Positive (GT). Set when the result is positive (and not zero).</td>
</tr>
<tr>
<td>2</td>
<td>Zero (EQ). Set when the result is zero.</td>
</tr>
<tr>
<td>3</td>
<td>Summary overflow (SO). This is a copy of the final state of XER[SO] at the completion of the instruction.</td>
</tr>
</tbody>
</table>

\[
\text{cmpw} \quad R1, \ R2 \quad ; \quad \text{== cmp} \ 0, \ 1, \ R1, \ R2
\]

; You may store results into another condition register
; for example, cmpw CR1, R1, R2

R1: 10 5 10
R2: 5 10 10
CR0: 010x 100x 001x

R1 > R2  R1 < R2  R1 == R2

How can you check <=, >=, and != ?
**Branch conditional**

```plaintext
bc   BO, BI, address

BO:  5 bits. See the next slide
BI:  5 bits specify which bit in the condition register (CR) to be tested

```

```plaintext
cmpw  R1, R2    ; set CR0: (LT, GT, EQ, 0)
bc    12, 2, equal ; go to equal if (R1 == R2)
       ; 12 = 0b01100
       ; bit 2 is EQ
       ; same as "beq    equal"

equal:
```
Count register (CTR)

- A 32-bit register that holds a loop count
- Can be decremented by using proper BO fields in conditional branch instructions
  - Conditional branch can also do extra computations on CTR
    - Controlled by Bit 2 in the BO field (the one in the middle)

Table 8-7. BO Operand Encodings

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000y</td>
<td>Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.</td>
</tr>
<tr>
<td>0001y</td>
<td>Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.</td>
</tr>
<tr>
<td>001zy</td>
<td>Branch if the condition is FALSE.</td>
</tr>
<tr>
<td>0100y</td>
<td>Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.</td>
</tr>
<tr>
<td>0101y</td>
<td>Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.</td>
</tr>
<tr>
<td>011zy</td>
<td>Branch if the condition is TRUE.</td>
</tr>
<tr>
<td>1000y</td>
<td>Decrement the CTR, then branch if the decremented CTR ≠ 0.</td>
</tr>
<tr>
<td>1001y</td>
<td>Decrement the CTR, then branch if the decremented CTR = 0.</td>
</tr>
<tr>
<td>111zz</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

z bits are ignored and should be cleared, as they may be assigned a meaning in a future version of the architecture. y bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.
Loop using GPR

; for (i = 0; i < 10; i++)
; {
}

main:
    xor R1, R1, R1
again:
    cmpwi R1, 10 ; which condition register?
    bge exit ; if (i >= 10) goto exit

    addi R1, R1, 1 ; i ++
    b again ; continue

exit:
Loop using CTR (1)

```assembly
main:
  xor  R1, R1, R1
  addi R1, R1, 10
  mtctr R1 ; set the count register
again:
  bc   18, 0, exit ; CTR = CTR - 1
                ; go to exit if CTR == 0
                ; bdz exit
  ...... ; loop body
  b     again ; continue
exit:

;   CTR = 10
;   while (-- CTR) do {
;     }
```
Loop using CTR (2)

main:
    li   R1, 10
    mtctr R1 ; set the count register
again:
    ...... ; loop body
    bdnz again ; CTR = CTR - 1
                ; go to again if CTR != 0
exit:

;    CTR = 10
;    do {
;         } while (-- CTR)

* bdnz is listed in Table F-4 (PE:page F-7)
Call functions using link register (LR)

main:
    or    R1, R2, R3
    bl    foo       ; call foo
    and   R1, R1, R2 ;

........

foo:
    add   R1, R1, R1 ; the address of and instruction
                     ; is stored in the link register
    blr   ; return
           ; branch to the link register

........

; Do not destroy the value in the link register if you want to use it later
Function calls with Link register

main:
    bl    bar
    nop

bar:
    mflr r22
    bl    foo
    mtlr r22
    ....
    blr

LR = main + 4

foo:
    ....
    blr

LR = bar + 8
More on function call

• Parameter passing
  – Register
    • Small number of parameters
    • Do not need the address of parameters
  – Stack

• Register preservation and usage
  – Save to memory the registers you will change
  – Restore registers on exit
Calling convention (Linux)

- R0: scratch registers
- R1: stack pointer
- R2: system reserved
- R3 – R10: parameter and result passing
  - R3 and R4 are also return values
- R11 – R12: scratch registers
- R13: global pointer to the small data area
- R14 – R31: global integer registers
  - R31 may be used as environment pointers
### Calling a function

- **Caller**
  - Save scratch registers if necessary
  - Prepare parameters
  - Call

- **Callee**
  - Save non-scratch registers to be changed
    - Save the registers on stack
    - Need to save stack information
  - Restore non-scratch registers before return
Integer exception register

- Summary overflow (SO), Overflow (OV), and Carry (CA)
- Affected by instructions such as `addo` and `addco`
- CA is set and used by `addc`, `adde`, and other instructions

<table>
<thead>
<tr>
<th>Field</th>
<th>SO</th>
<th>OV</th>
<th>CA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0000_0000_0000_0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R/W</td>
<td>R/W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>—</th>
<th>BCNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0000_0000_0000_0000</td>
<td></td>
</tr>
<tr>
<td>R/W</td>
<td>R/W</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 4-2. XER Register*
Using the carry bit

```
addc R1, R2, R3
addze R4, R4
```

R2: 1111 0000 0000 0000 0000 0000 0000 0000
R3: 1000 0000 0000 0000 0000 0000 0000 0000
R1: 0111 0000 0000 0000 0000 0000 0000 0000
XER[CA]: 1

New R4 = Old R4 + 0 + 1

Can you do similar things in C?
mfspr and mtspr instructions

- Move from special-purpose registers (to GPR)
- Move to special-purpose registers (from GPR)

Mnemonics !!!

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operand</th>
<th>Instruction</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfxer</td>
<td>R1</td>
<td>mfspr</td>
<td>R1, 1</td>
</tr>
<tr>
<td>mflr</td>
<td>R1</td>
<td>mfspr</td>
<td>R1, 8</td>
</tr>
<tr>
<td>mfctr</td>
<td>R1</td>
<td>mfspr</td>
<td>R1, 9</td>
</tr>
<tr>
<td>mtxer</td>
<td>R1</td>
<td>mtspr</td>
<td>1, R1</td>
</tr>
<tr>
<td>mtlr</td>
<td>R1</td>
<td>mtspr</td>
<td>8, R1</td>
</tr>
<tr>
<td>mtctr</td>
<td>R1</td>
<td>mtspr</td>
<td>9, R1</td>
</tr>
</tbody>
</table>
Simplified mnemonics

No-op:

\[
\text{nop} \\
\text{ori} \quad \text{R0, R0, 0}
\]

Load immediate into registers:

\[
\text{li} \quad \text{R1, IMME} \\
\text{addi} \quad \text{R1, 0, IMME}
\]

NOTE: IMME has only 16 bits and will be sign extended to 32 bits

Load immediate into registers and shift to left by 16:

\[
\text{lis} \quad \text{R1, 0x1FB} \\
\text{addis} \quad \text{R1, 0, 0x1FB}
\]

Move registers:

\[
\text{mr} \quad \text{R1, R2} \\
\text{or} \quad \text{R1, R2, R2}
\]
Example: Embedded instructions in C code (1)

```c
long gvar = 5;
void func(register int *a)
{
    int c;
    asm
    {
        Label1:    // define label
            lwz r6, 0(a)     // load *a to r6
            stw r6, c
            lwz r6, c
            lwz r6, c(SP)   // access local variable
            lwz r7, gvar    // access global variables
            add r6, r6, r7  // *a = *a + gvar
            stw r6, 0(r5)   // save *a
    }
}
```
Example: Embedded instructions in C code (2)

```c
asm func1(register int *a)
{
    int c;

    // Create a stack frame for a function when it calls
    // other functions, declares non-register arguments
    // or local variables

    fralloc
    lwz r6, 0(a)   // load *a to r6
    stw r6, c
    lwz r7, gvar   // access global variables
    add r6, r6, r7 // *a = *a + gvar
    stw r6, 0(r5)  // save *a
    frfree         // free the stack frame
    blr
}
```
Example: Embedded instructions in C code (3)

```c
asm func1(register int *a)
{
    long myVar;
    long myArray[1];
    Rect myRectArray[3];

    fralloc
    l1: lwz r3,myVar(SP)
    la r3,myVar(SP)
    lwz r3,myRect.top
    @1 lwz r3,myArray[2](SP)
    lwz r3,myRectArray[2].top
    lbz r3,myRectArray[2].top+1(SP)
    frfree // free the stack frame
    blr
} // more on page 80 of the CW manual.
```
Implement a function with assembly code

• In C file:

    void my_strcpy(register int *d, register int *s);

• In assembly file:

    .global my_strcpy
    .text
    my_strcpy:
        subi r3, r3, 1 ;
        subi r4, r4, 1 ;
    next:
        lbzu r5, 1(r4) ; read a byte from s
        stbu r5, 1(r3) ; store the byte to d
        cmpwi r5, 0 ; if it is not 0, continue to copy
        bne next
        blr
Memory map

OS data and exception handler

Data and code in DRAM

16 KB memory resource decided by the upper 16 bits of Internal Memory Map Register (IMMR)

Other purposes
Reading IMMR in C

unsigned int GetIMMR(void)
{
    asm {
        mfspr r3, IMMR
        andis. r3, r3, 0xFFFF
    }
}
Readings

- PE(Chapter 2): Register set
- PE(Chapter 4): Addressing mode and instruction summary
- PE(Chapter 8): Complete instruction list
- PE(Appendix F): Mnemonics
- CodeWarrior Development Tools: Assembler References
  - Assembly Language Syntax
  - Directives
  - Macros

PE: the program environments manual for 32-bit Implementations of the PowerPC architecture.