Lecture 7

# Computer Architecture
## (Spring 2020)

Instruction Set Principles

Dr. Duo Liu (刘铎)
Office: Main Building 0626
Email: liuduo@cqu.edu.cn

# MIPS

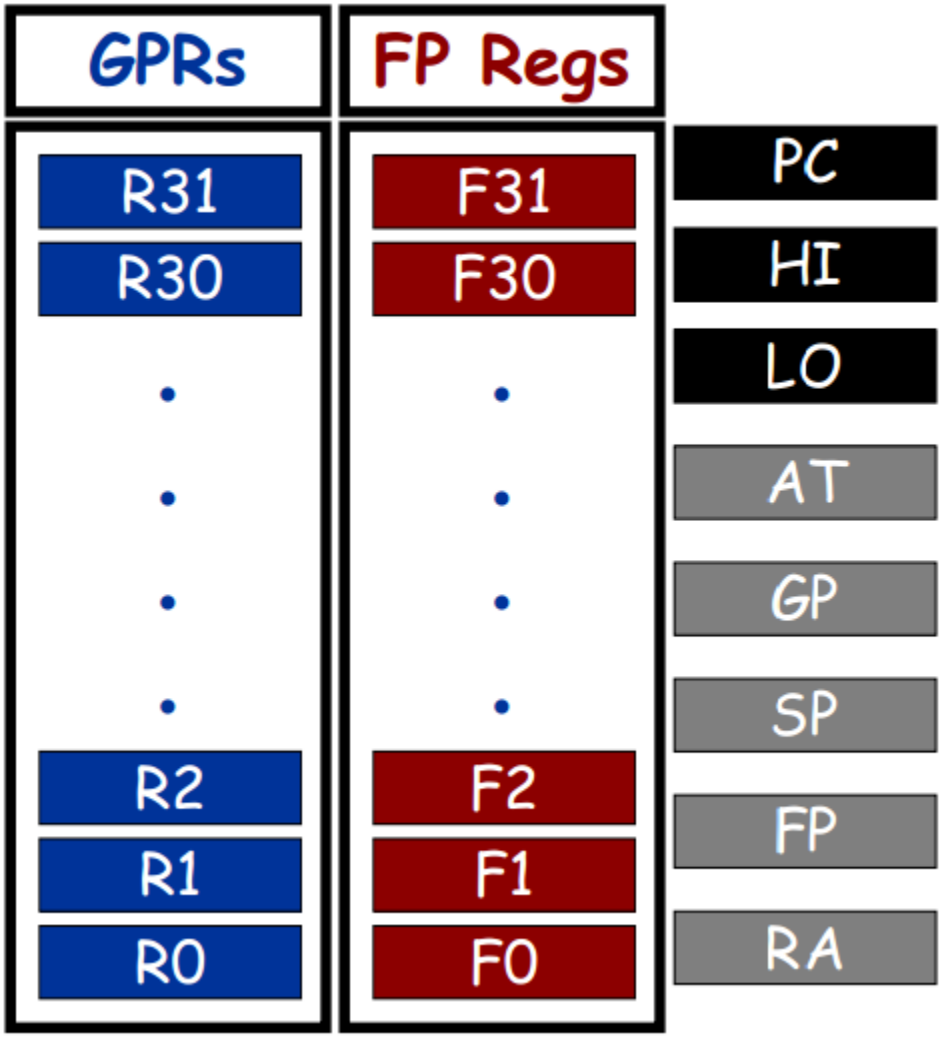
- 1981: Stanford MIPS Computer, a "Microprocessor without Interlocked Pipeline Stages" [Hennessy81]
  - no HW to stall the pipeline (handling dependencies is compiler's job)
- 1984: Hennessy founds MIPS Computer System
  - R2000 & R3000 first products ( with interlock in HW! )
- 1991: MIPS releases the 64-bit R4000
  - SGI buys MIPS which becomes the division MIPS Technologies
- 1999: MIPS-Technologies so successful that SGI spins off it
  - two products MIPS32, MIPS64
  - major revenues portion from licensing the design
  - estimated 1/3 of the produced RISCs is a MIPS-based design
    - almost 100 million of MIPS manufactured in 2002
    - Used in products from ATI Technologies, Broadcom, Cisco, NEC, Nintendo
      SGI, Sony, Texas Instruments, Toshiba…
- MIPS64 is used throughout the textbook to illustrate the ILP techniques

# Before Introducing MIPS64: What We Expect?

- Architecture?
  - register-to-register (load-store)
- Registers?
  - “many” general purpose registers and dedicated FP registers
- Addressing modes support?
  - displacement (address-offset of 12 to 16 bits)
  - using PC-relative addressing, branch +/- 2^15 words from PC
  - immediate (size 8-16 bits)
  - register indirect
- Data sizes & types?
  - Integers (8,16,32,64 size) and FP (64)
- Instructions types?
  - strong support for simple pervasive instructions
    - load, store, add, sub, move register-register, and shift
    - compare equal/not equal/less, branch, jump, call, return
- Encoding?
  - fixed for performance rather than variable for code size

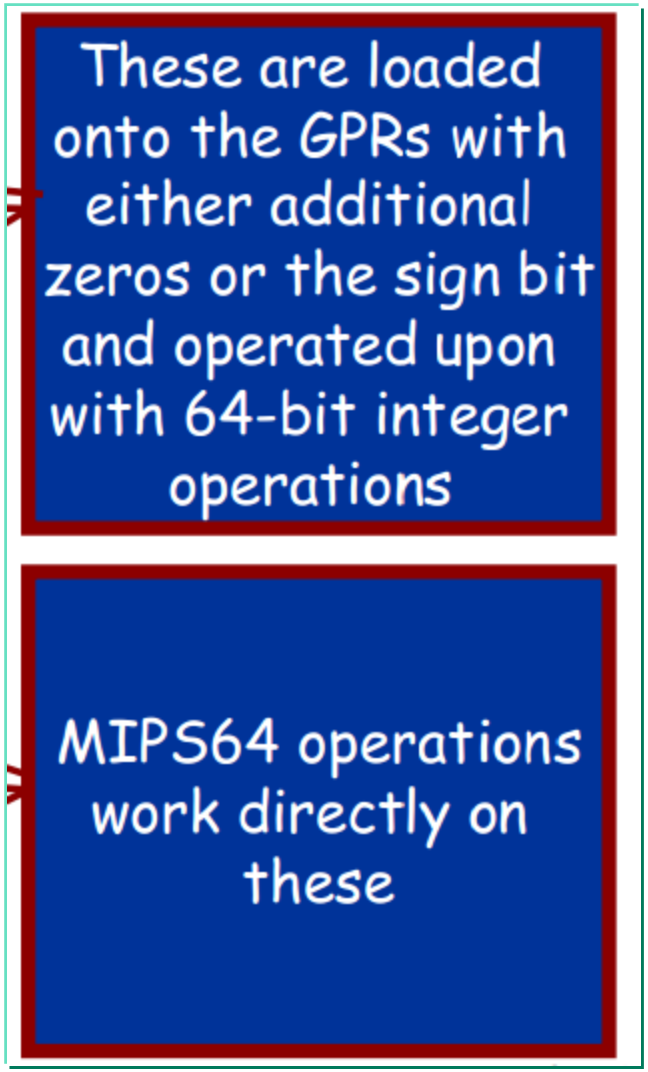
# MIPS - Registers

- 32 64-bit GPRs
  - R0 is hardwired to zero
  - AT, GP, SP, FP, RA are conventionally mapped to R1, R28, R29, R30, R31
- 32 Floating Point Registers holding either
  - 32 single-precision values (32 bits)
    - half of the FPR is not used
  - 32 double-precision values (64 bits)
- Special registers
  - PC – program counter
  - HI, LO
    - integer multiply result (higher and lower word)
    - integer divide result (remainder and quotient)



# MIPS – Data Types

- Integer data
  - 8-bit bytes
  - 16-bit half words
    - present in C and popular in programs like operating systems (concerned about size of data structures)
    - also even more popular if Unicode becomes more pervasive
  - 32-bit words
  - 64-bit double words
- Floating Point
  - 32-bit single precision
    - Same motivations as for the 16-bit half words
  - 64-bit double precision



# MIPS – Addressing Modes

- The hardware only support two addressing modes
  - **immediate** (16-bit)

    DADDI R3, R2, #7 ; Regs[R3] ←Regs[R2] + 7
  - **displacement** (16-bit)

    LD R3, 100(R1) ; Regs[R3] ←Mem[100+Regs[R1]]

- Two other modes supported "indirectly"
  - **register indirect** (placing 0 in the 16-bit displacement field)

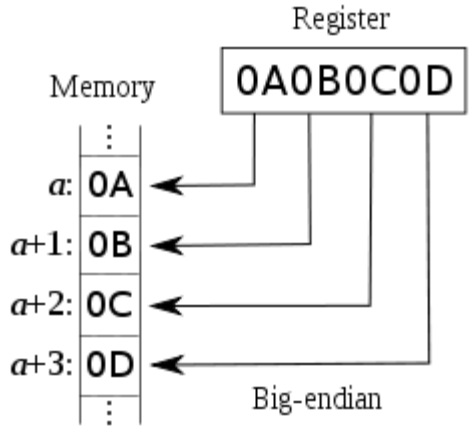
    LD R3, 0(R1) ; Regs[R3] ←Mem[Regs[R1]]
  - **direct addressing** (using 0 as the base register)

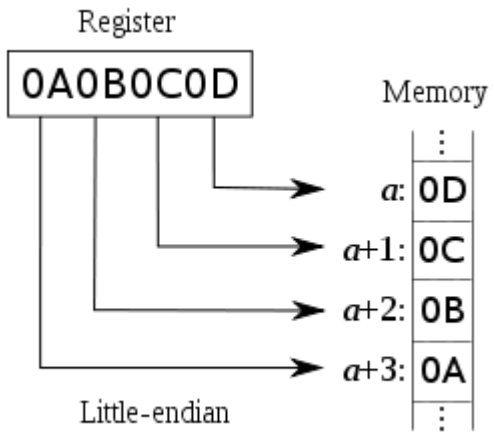
    LD R3, 1001(R0) ; Regs[R3] ←Mem[1001]

- Memory is byte-addressable with a 64-bit address

# Interpreting Memory Addresses

- Memory word is addressed by the byte at the lowest address
- Big Endian
  - Most significant byte is at the lowest address
  - "big end first"
  - Ex: Motorola 68000, SPARC, IBM 360
- Little Endian
  - Least significant byte is at the lowest address
  - "little end first"
  - Ex: Intel x86, AMD64, VAX




Some architectures can be configured in both ways: ARM, MIPS, IA64, PowerPC

# MIPS – Instruction Format and Layout

- MIPS instructions fall into 5 classes:
  - Arithmetic/logical/shift/comparison
  - Control instructions (branch and jump)
  - Load/store
  - Other (exception, register movement to/from GP registers, etc.)

- Three instruction encoding formats:
  - R-type (6-bit opcode, 5-bit rs, 5-bit rt, 5-bit rd, 5-bit shamt, 6-bit function code)

| 31-26 | 25-21 | 20-16 | 15-11 | 10-6 | 5-0 |
|---|---|---|---|---|---|
| *opcode* | *rs* | *rt* | *rd* | shamt | function |

  - I-type (6-bit opcode, 5-bit rs, 5-bit rt, 16-bit immediate)

| 31-26 | 25-21 | 20-16 | 15-0 |
|---|---|---|---|
| *opcode* | *rs* | *rt* | *imm* |

  - J-type (6-bit opcode, 26-bit pseudo-direct address)

| 31-26 | 25-0 |
|---|---|
| *opcode* | *pseudodirect jump address* |

# Example Instructions

- ADD $2, $3, $4
  - R-type A/L/S/C instruction
  - Opcode is 0's, rd=2, rs=3, rt=4, func=000010
  - 000000 00011 00100 00010 00000 000010

- JALR $3
  - R-type jump instruction
  - Opcode is 0's, rs=3, rt=0, rd=31 (by default), func=001001
  - 000000 00011 00000 11111 00000 001001

- ADDI $2, $3, 12
  - I-type A/L/S/C instruction
  - Opcode is 001000, rs=3, rt=2, imm=12
  - 001000 00011 00010 0000000000001100

# Example Instructions

- BEQ $3, $4, 4
  - I-type conditional branch instruction
  - Opcode is 000100, rs=00011, rt=00100, imm=4 (skips next 4 instructions)
  - 000100 00011 00100 0000000000000100
- SW $2, 128($3)
  - I-type memory address instruction
  - Opcode is 101011, rs=00011, rt=00010, imm=0000000010000000
  - 101011 00011 00010 0000000010000000
- J 128
  - J-type pseudodirect jump instruction
  - Opcode is 000010, 26-bit pseudodirect address is 128/4 = 32
  - 000010 00000000000000000000100000

# MIPS – Some Load & Store Instructions

- Load Double Word

`LD R1, 30(R2)` ;Regs[R1] $\leftarrow_{64}$ Mem[30 + Regs[R2]]

- Load Double Word (direct addressing using zero as base reg)

`LD R1, 30(R0)` ;Regs[R1] $\leftarrow_{64}$ Mem[30 + 0]

- Load Byte

`LB R1, 40(R3)` ;Regs[R1]$\leftarrow_{64}$ (Mem[40 + Regs[R3]]$_0$) $^{56}$ ## Mem[40 + Regs[R3]]

- Load Byte Unsigned

`LBU R1, 40(R3)` ;Regs[R1] $\leftarrow_{64}$ $0^{56}$ ## Mem[40 + Regs[R3]]

- Load FP single

`L.S F0, 50(R3)` ;Regs[F0] $\leftarrow_{64}$ Mem[50 + Regs[R3]] ## $0^{32}$

- Load FP double

`L.D F0, 50(R2)` ;Regs[F0] $\leftarrow_{64}$ Mem[50 + Regs[R2]]

- Store Half Word

`SH R3, 502(R2)` ;Mem[502 + Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48..63}$

- Store FP single

`S.D F0, 40(R3)` ;Mem[40 + Regs[R3] $\leftarrow_{32}$ Regs[F0]]$_{0..31}$

# MIPS – Some Arithmetic/Logic Instructions

- Double Word Add

**DADD R1, R2, R3** ;Regs[R1] ← Regs[R2] + Regs[R3]

- Double Word Add Immediate Unsigned

**DADDIU R1, R2, #3** ;Regs[R1] ← Regs[R2] + 3

- Double Word Shift Left Logical

**DSLL R1, R2, #5** ;Regs[R1] ← Regs[R2] << 5

- Set on Less Than

SLT R1, R2, R3 ; if (Regs[R2] < Regs[R3])
Regs[R1] ← 1 else Regs[R1] ← 0

- Multiply and Add Word to HI,LO registers

**MADD R1, R2** ; (LO,HI) ← Regs[R1] x Regs[R2] + (LO,HI)

- Loading a Constant (mnemonic: **LI**)

**LI R1, #3** ;Regs[R1] ← 3

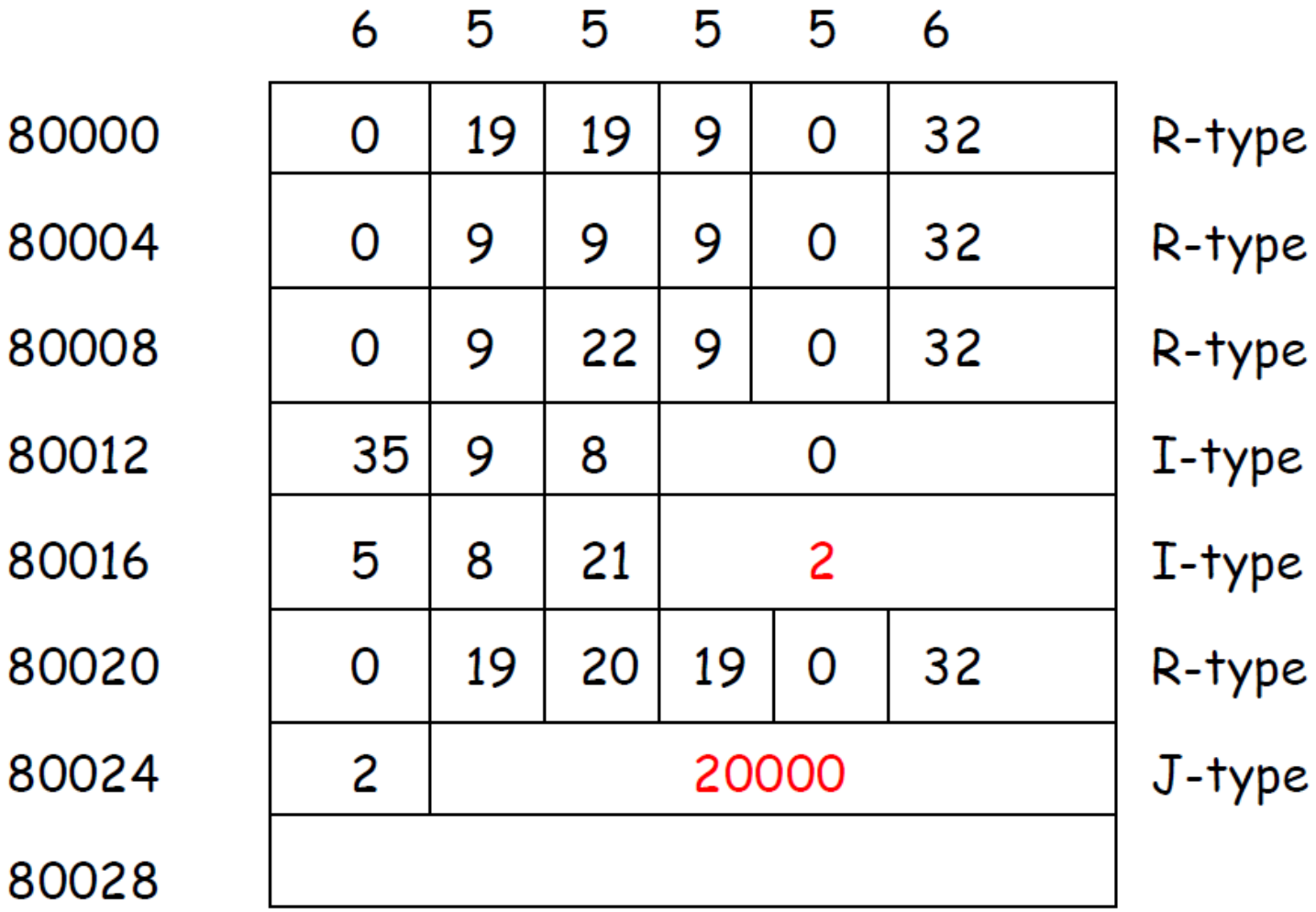
- Register-Register Move (mnemonic: **MOV**)

**MOV R1,R2** ;Regs[R1] ← Regs[R2]

# EXAMPLE

```
Loop:   add  $t1, $s3, $s3    # starts from 80000
        add  $t1, $t1, $t1
        add  $t1, $t1, $s6
        lw   $t0, 0($t1)
        bne  $t0, $s5, Exit
        add  $s3, $s3, $s4
        j    Loop
Exit:
```

# EXAMPLE


6 5 5 5 5 6
80000 0 19 19 9 0 32 R-type
80004 0 9 9 9 0 32 R-type
80008 0 9 22 9 0 32 R-type
80012 35 9 8 0 I-type
80016 5 8 21 2 I-type
80020 0 19 20 19 0 32 R-type
80024 2 20000 J-type
80028

# MIPS – Some Arithmetic/Logic Instructions

- Suppose register R3 has the binary number

1111 1111 1111 1111 1111 1111 1111 1111

- and register R4 has the binary number

0000 0000 0000 0000 0000 0000 0000 0001

- What are the values of R1 and R2 after this?

**SLT R1 R3 R4**

**SLTU R2 R3 R4**

- Answer:

**R1 = 1, R2 = 0**

- because

R3 = -1 (as an integer) and =4,294,967,295 (as unsigned)

R4 = 1 in both cases

# MIPS – Some Control-Flow Instructions

- Jump (28-bit target address is 26-bit name shifted left twice)

**J name** ; PC ← $PC_{64..28}$ ## name ## $0^2$

- Jump and link (using the return address register RA≡R31)

**JAL name** ;Regs[R31] ← PC+8; PC ← $PC_{64..28}$ ## name ## $0^2$

- Jump and link register

**JALR R2** ;Regs[R31] ← PC+8; PC ← Regs[R2]

- Jump register

**JR R3** ;PC ← Regs[R3]

- Branch equal (zero) (compilers combined this with SLT, SLTI in order to build branch-on-less conditions)

**BEQ R4, R0, name** ;if (Regs[R4]==0) PC ← PC + signExt{name ## $0^2$ }

- Branch not equal

**BNE R3,R4,name** ;if(Regs[R3]!= Regs[R4]) PC ← PC + signExt{name ## $0^2$ }

- Conditional move if zero (support conversion of a simple branch into a conditional arithmetic instruction)

**MOVZ R1,R2,R3** ;if(Regs[R3]==0) Regs[R1] ← Regs[R2]

# Review: MIPS-Jump Instruction Example

```
/* Assume loop stored at
   0000 A000 6BC4 0308 */
Loop: SLL R9,R19,2
      ADD R9,R9,R22
      LW  R8, 0(R9)
      BNE R8, R21,exit
      ADDI R19,R19,1
      J Loop
Exit: LW R11, 0(R18)
```

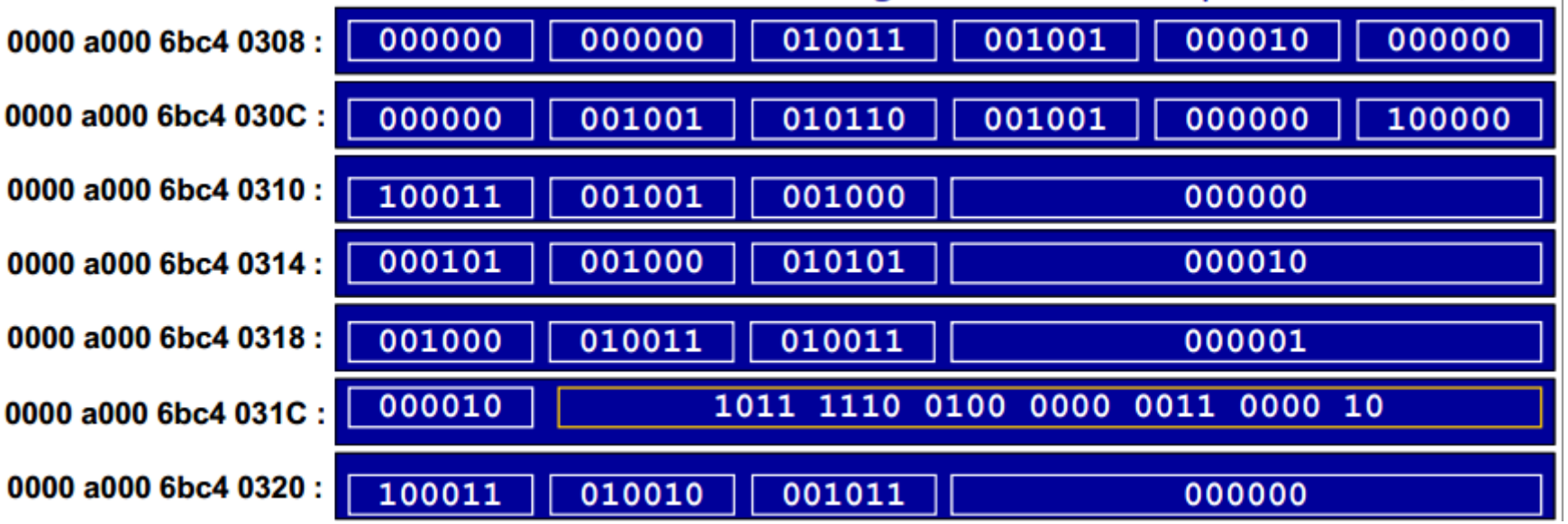
- `j loop` : [PC-region addressing] the target address effectively encoded is `2F9 00C2` that shifted-left by two and composed with the upper 36 bits of the PC of the following instruction gives `0000 a000 6bc4 0308`
- `BNE R8, R21, exit` : [PC-relative addressing] the offset is 2 (it gets summed to the PC of the following instruction to point to Exit)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0000 a000 6bc4 0308 : | SLL | 0 | R19 | R9 | 2 | sll |
| 0000 a000 6bc4 030C : | ADD | R9 | R22 | R9 | 000000 | add |
| 0000 a000 6bc4 0310 : | LW | R9 | R8 | 0 | | |
| 0000 a000 6bc4 0314 : | BNE | R8 | R21 | 2 | | |
| 0000 a000 6bc4 0318 : | ADDI | R19 | R19 | 1 | | |
| 0000 a000 6bc4 031C : | JUMP | 2F9 00C2 = (bc4 0308 / 4) | | | | |
| 0000 a000 6bc4 0320 : | LW | R18 | R11 | 0 | | |

# Review: MIPS-Jump Instruction Example

```
/* Assume loop stored at
   0000 A000 6BC4 0308 */
Loop: SLL R9,R19,2
      ADD R9,R9,R22
      LW  R8, 0(R9)
      BNE R8, R21,exit
      ADDI R18,R18,1
      J Loop
Exit: LW R11, 0(R18)
```

- `j loop` : [PC-region addressing] the target address effectively encoded is `2F9 00C2` that shifted-left by two and composed with the upper 36 bits of the PC of the following instruction gives `0000 a000 6bc4 0308`
- `BNE R8, R21, exit` : [PC-relative addressing] the offset is 2 (it gets summed to the PC of the following instruction to point to Exit)



# MIPS – Floating-Point Instructions

- Floating Point Absolute Value (Double Precision)

```
ABS.D F1, F2          ;F1 ← abs(F2)
```

- Floating Point Add (Single Precision)

```
ADD.S F3, F1, F2 ;Regs[F3] ← Regs[F1] + Regs[F2]
```

- Floating Point Add (Double Precision)

```
ADD.D F3, F1, F2 ;Regs[F3] ← Regs[F1] + Regs[F2]
```

- Floating Point Divide (Double Precision)

```
DIV.D F3, F1, F2 ;Regs[F3] ← Regs[F1] / Regs[F2]
```

- Floating Point Square Root (Double Precision)

```
SQRT.D F3, F1         ;Regs[F3] ← sqrt(Regs[F1])
```

- Floating Point Multiply Add (Single Precision)

```
MADD.S F3, F1, F2, F4     ;Regs[F3] ←(Regs[F2] x Regs[F4)] +
                          + Regs[F1]
```

# Real World Instruction Sets

| Arch | Type | # Oper | # Mem | Data Size | # Regs | Addr Size | Use |
|---|---|---|---|---|---|---|---|
| Alpha | Reg-Reg | 3 | 0 | 64-bit | 32 | 64-bit | Workstation |
| ARM | Reg-Reg | 3 | 0 | 32/64-bit | 16 | 32/64-bit | Cell Phones, Embedded |
| MIPS | Reg-Reg | 3 | 0 | 32/64-bit | 32 | 32/64-bit | Workstation, Embedded |
| SPARC | Reg-Reg | 3 | 0 | 32/64-bit | 24-32 | 32/64-bit | Workstation |
| TI C6000 | Reg-Reg | 3 | 0 | 32-bit | 32 | 32-bit | DSP |
| IBM 360 | Reg-Mem | 2 | 1 | 32-bit | 16 | 24/31/64 | Mainframe |
| x86 | Reg-Mem | 2 | 1 | 8/16/32/ 64-bit | 4/8/24 | 16/32/64 | Personal Computers |
| VAX | Mem-Mem | 3 | 3 | 32-bit | 16 | 32-bit | Minicomputer |
| Mot. 6800 | Accum. | 1 | 1/2 | 8-bit | 0 | 16-bit | Microcontroler |

# Why the Diversity in ISAs?

Technology Influenced ISA

- Storage is expensive, tight encoding important
- Reduced Instruction Set Computer
  - Remove instructions until whole computer fits on die
- Multicore/Manycore
  - Transistors not turning into sequential performance

Application Influenced ISA

- Instructions for Applications
  - DSP instructions
- Compiler Technology has improved
  - SPARC Register Windows no longer needed
  - Compiler can register allocate effectively

# Quize 2

1. H&P5th Page A-47 A.1

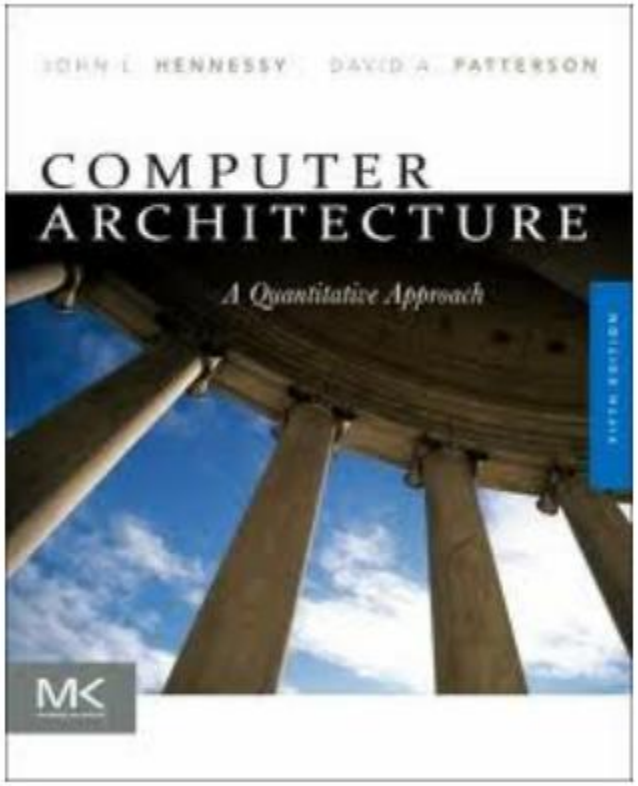
2.
[Self-Study]：
The value represented 0x(0123 4567 89AB CDEF) is to be stored in an aligned 64bit double word. Choose an address from 0x00002 to 0x00012 as the starting address, and store them using Little Endian byte order.

# Assigned Readings

- Computer Architecture – A Quantitative Approach by
  John Hennessy
  - Stanford University

  Dave Patterson
  - UC Berkeley

  Fifth Edition - 2012
  Morgan Kaufmann (Elsevier)




- Read Sections

Appendix A.1-A.7, A.9-A.11

# Homework #2

H&P 5th

Page A-47: A.3, A.7, A.9

Due 2020/04/09