UNIVERSITY OF SOUTH CAROLINA

Review: MIPS Instructions

- MIPS instruction types:
  - Arithmetic/logical/shift/comparison
  - Control instructions (branch and jump)
  - Load/store

- 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*

Your CPU: MIPS Instructions

- Arithmetic R-type: add, addu, sub, subu
- Arithmetic I-type: addi, addiu
- Comparison R-type: slt, sltu
- Comparison I-type: slti, sltiu
- Logical R-type: and, or, nor, xor
- Logical I-type: andi, ori, xori
- Shift R-type: sll, sllv, srl, srlv, sra, srav
- Load/Store I-type: lui, lw, lh, lhu, lb, lbu, sw, sh, sb
- Branch I-type:
  - beq, bne, bgez, bgezal, bgtz, blez, bltzal, bltz
- Jump J-type: j, jal
- Jump R-type: jr, jalr

Review: MIPS Addressing Modes

- MIPS addresses register operands using 5-bit field
  - Example: **`ADD $2, $3, $4`**
- Immediate addressing
  - Operand is held as constant (literal) in instruction word
  - Example: **`ADDI $2, $3, 64`**
- MIPS addresses branch targets as signed instruction offset
  - relative to next instruction ("PC relative")
  - in units of instructions (words)
  - held in 16-bit offset in I-type
  - Example: **`BEQ $2, $3, 12`**

Review: MIPS Addressing Modes (con't)

- MIPS addresses jump targets as register content
  - Example: **`JR $31`**

– …or as 26-bit "pseudo-direct" address
  - Example: **`JR $31, J 128`**

- MIPS addresses load/store locations
  - base register + 16-bit signed offset (byte addressed)
    - Example: **`LW $2, 128($3)`**
  - 16-bit direct address (base register is 0)
    - Example: **`LW $2, 4092($0)`**
  - indirect (offset is 0)
    - Example: **`LW $2, 0($4)`**

Review: 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

Review: 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

UNIVERSITY OF SOUTH CAROLINA

Review: MIPS Registers

- 32 general purpose *integer* registers
  - Some have special purposes
  - These are the only registers the programmer can directly use
    - $0 => constant 0
    - $1 => $at (reserved for assembler)
    - $2,$3 => $v0,$v1 (function return value)
    - $4-$7 => $a0-$a3 (function arguments)
    - $8-$15 => $t0-$t7 (temporary values)
    - $16-$23 => $s0->$s7 (function local variables)
    - $24, $25 => $t8, $t9 (temporary values)
    - $26,$27 => $k0, $k1 (reserved for OS kernel)
    - $28 => $gp (pointer to global area)
    - $29 => $sp (stack pointer)
    - $30 => $fp (frame pointer)
    - $31 => $ra (function return address)

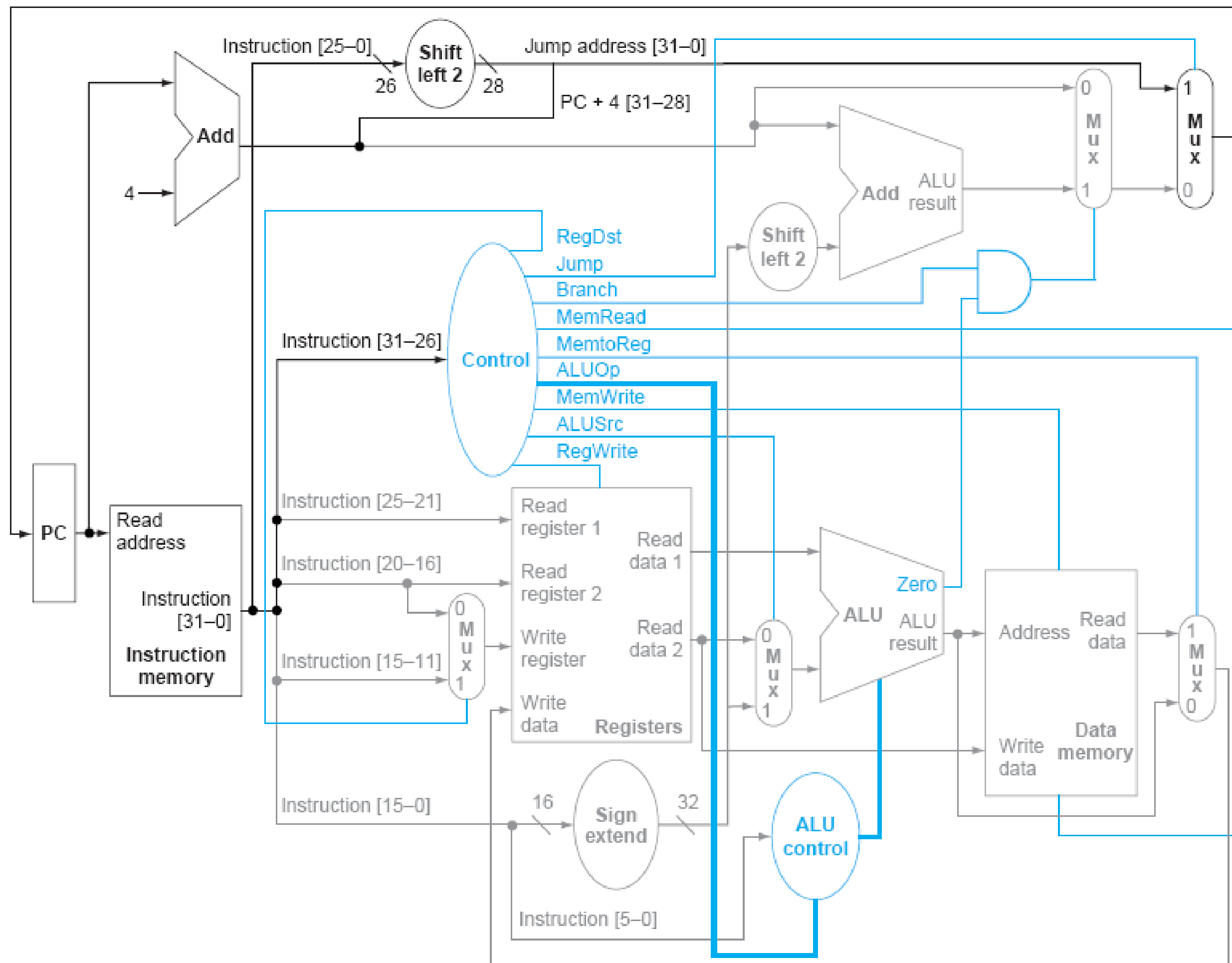
- Program counter (PC) contains address of next instruction to be executed

Review: MIPS Code Example

```
for (i=0;i<n;i++) a[i]=b[i]+10;
```

```
        li $s0,0              # i=0
        lw $s1,n              # load n
        sll $s1,$s1,2         # n = n * 4
loop:   bge $s0,$s1,exit      # check !(i < n)
        lw $s3,b($s0)         # load b
        addi $s3,$s3,10       # add 10
        sw $s3,a($s0)         # store to a
        addi $s0,$s0,4        # i += 4
        j loop                # loop
```

Review: MIPS Datapath with Jump



CPU Project

- Goal: design a pipelined processor that can execute all the instructions listed on slide 3
  - Use on-chip memory for program
    - Word addressed
  - Use on-chip memory for data
    - Word addressed but with byte-enables
  - Use three stage pipeline
    - Fetches need one cycle
    - Decode, execute, load/store needs one cycle
    - Register write back needs one cycle

CPU Project

- Tutorials 15-17 will get you started:
  - Top-level design, including processor and instruction/data memories
  - Inside processor design:
    - Instance register file, program counter and incrementer, control unit, ALU, branch target ALU
    - Implement load instructions
- You will be responsible for:
  - R-type and I-type ALU instructions
  - Store instructions
  - Branch instructions
  - Jump instructions

Top-Level Design

- Need two separate memories for instructions and data
  - Each has different characteristics

FPGA

Instruction memory

CLOCK_27

KEY(0)

clk

rst

CPU

32 bits

32-bit BRAM

Initialization??

8 bits

32-bit BRAM

Initialization??

Data memory

CPU Interface

clk

rst

InstructionIn(31:0)

DataIn(31:0)

InstAddr(9:0)

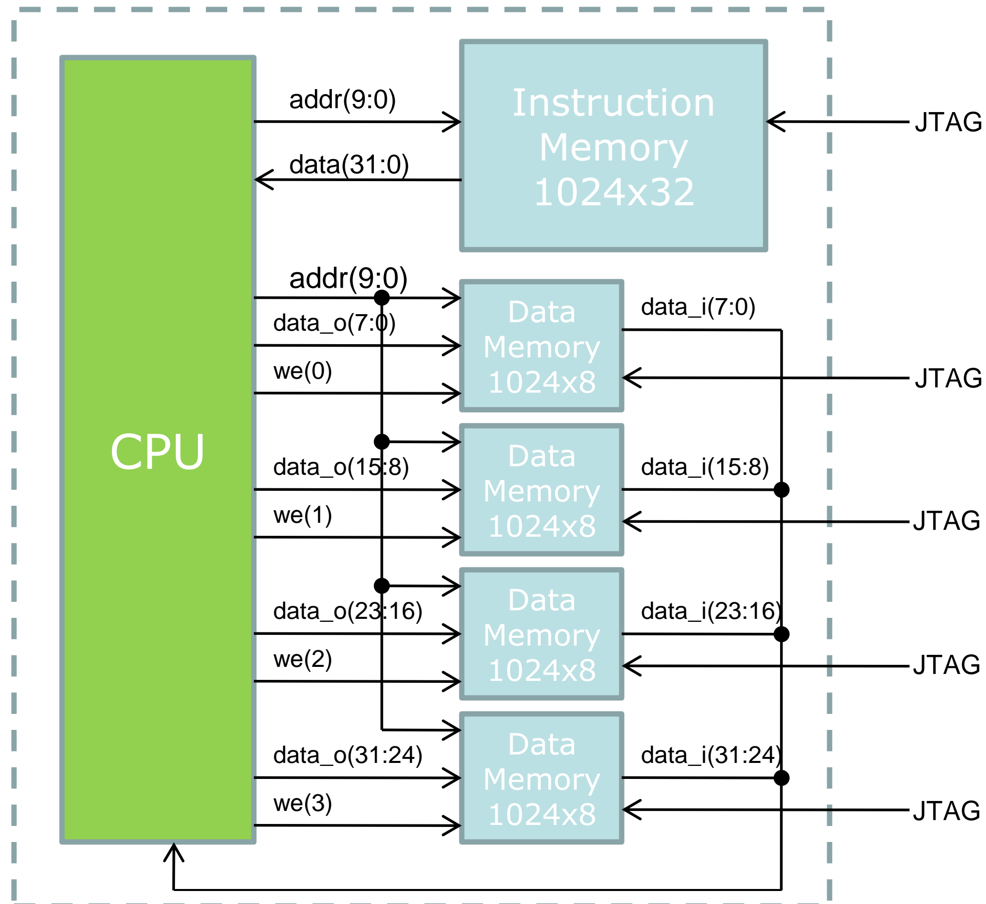
DataAddr(9:0)

DataOut(31:0)

DataWE(3:0)

“MyComputer Design”

FPGA



Synchronous Reads

- On-chip memory with synchronous reads

clk

addr: A1 A2 A3 A4 A5 A6 A7 A8

we

dataout: ? D1 D2 D3 D4 D5 E5 D7

datain: E5

CPU Design

- Loads have a one-cycle latency
  - Need to separate MEM and WB
- Fetches have a one-cycle latency
  - Need to separate FETCH and DECODE

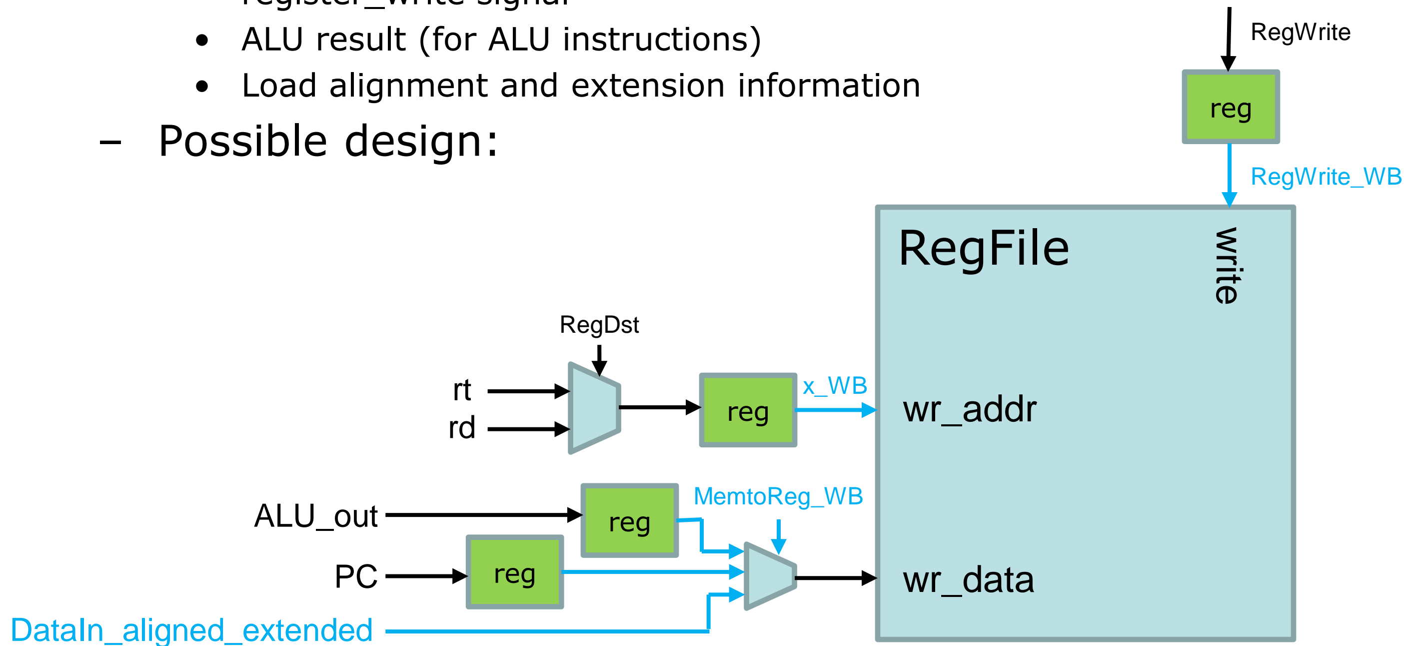
- Solution:
  - Three-stage pipeline:
    - FETCH, DECODE/EX/MEM, WB
    - F E W
  - “Register” all WB signals
  - Leads to control hazard
    - Must flush FETCH for taken branches
    - Zero-out control signals in if branch is taken in previous cycle

Three Stage Pipeline

cycle	0	1	2	3	4	5	6	7
add	F	E	W					
add		F	E	W				
branch (t)			F	E	W			
fall through				F	X	X		
target					F	E	W	
branch(nt)						F	E	W

Notes

- Some control signals and intermediate values need to be "delayed" for use in the write-back stage:
  - Add "_WB" suffix to these
  - Includes:
    - rt and rd field from instruction
    - register_write signal
    - ALU result (for ALU instructions)
    - Load alignment and extension information
  - Possible design:



Alignment

- Memory is a one-dimensional array of bytes
- Is “byte addressed”
- Registers are an array of 32-bit words

memory

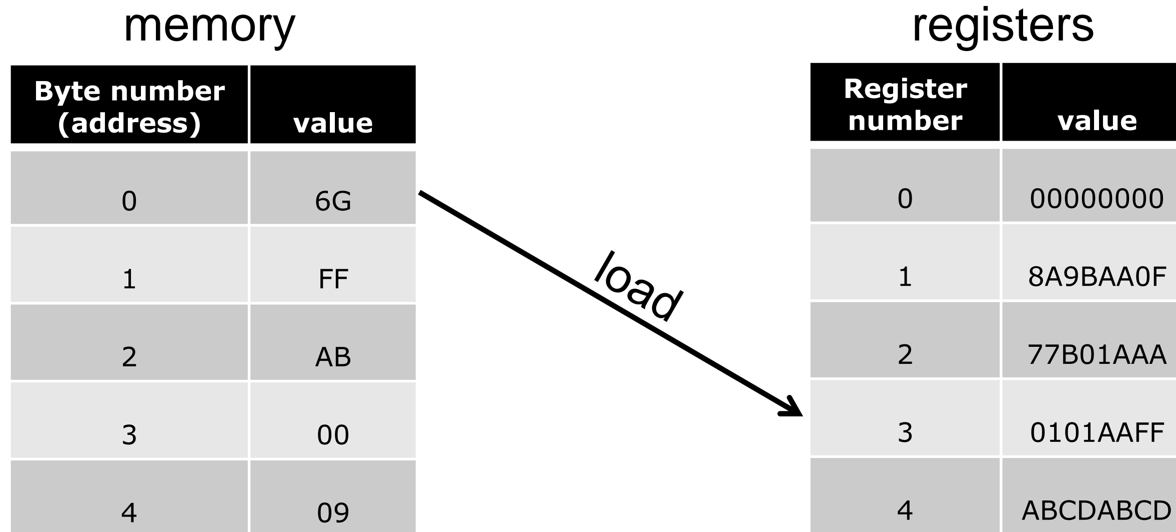
Byte number (address)	value
0	6G
1	FF
2	AB
3	00
4	09

registers

Register number	value
0	00000000
1	8A9BAA0F
2	77B01AAA
3	0101AAFF
4	ABCDABCD

Alignment

- Problem:
  - What happens when I LOAD address 0 into register 3?



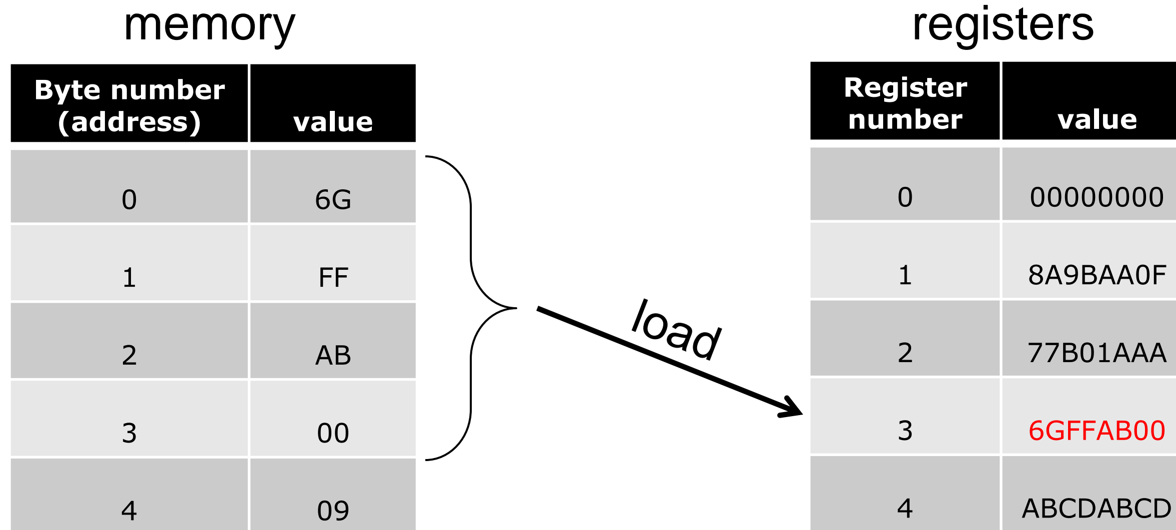
Byte number (address)	value
0	6G
1	FF
2	AB
3	00
4	09

Register number	value
0	00000000
1	8A9BAA0F
2	77B01AAA
3	0101AAFF
4	ABCDABCD

??????????????????????

Alignment

- Answer:
  - Memory provides FOUR bytes STARTING with address 0
  - In other words, we get addresses 1, 2 and 3 also!!!



memory

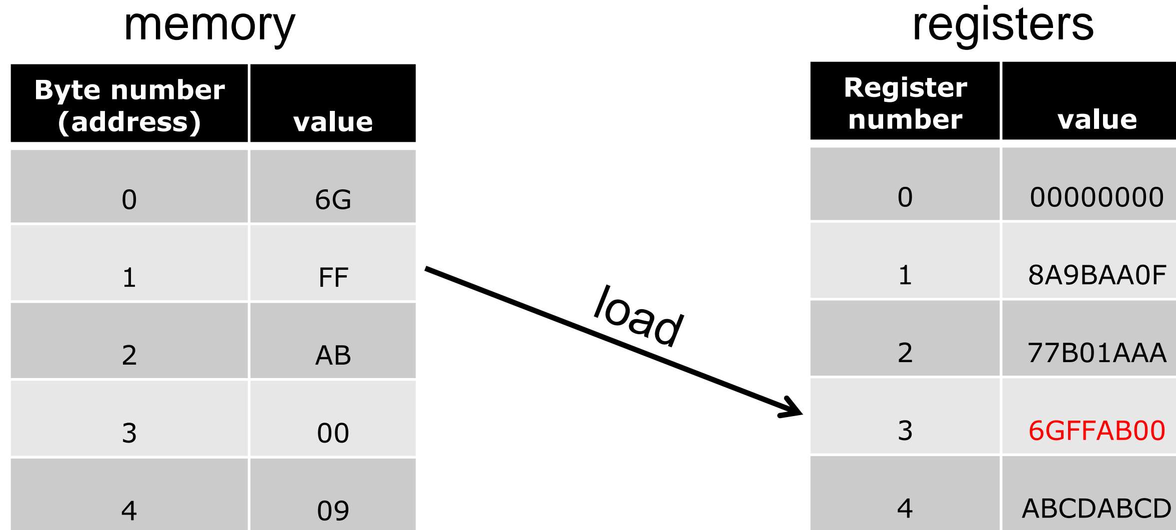
Byte number (address)	value
0	6G
1	FF
2	AB
3	00
4	09

registers

Register number	value
0	00000000
1	8A9BAA0F
2	77B01AAA
3	6GFFAB00
4	ABCDABCD

Alignment

- Problem:
  - What happens when I LOAD address 1 into register 3?
  - I already got the contents of address 1, 2, and 3 from last time!

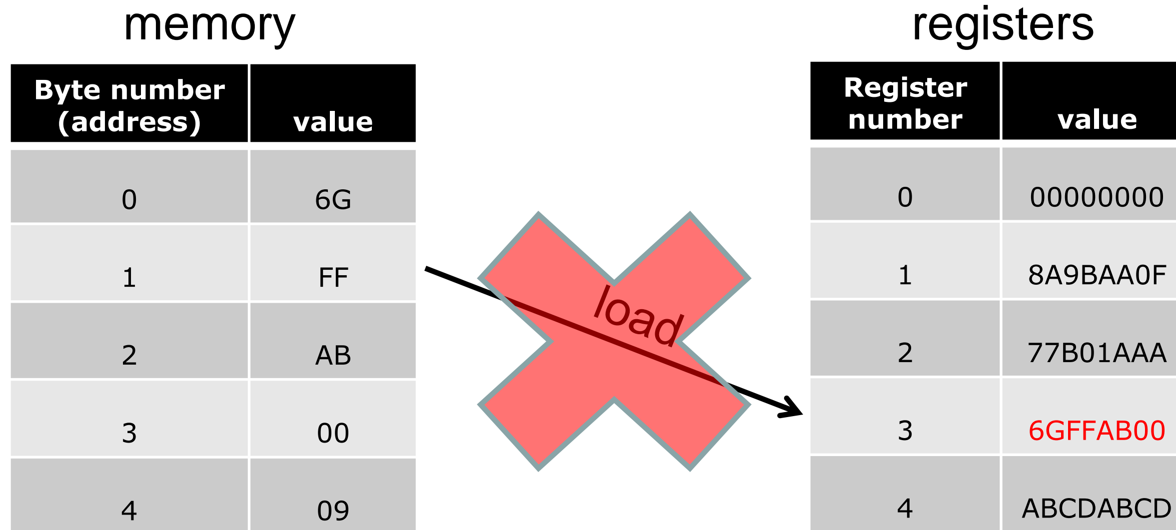


Byte number (address)	value
0	6G
1	FF
2	AB
3	00
4	09

Register number	value
0	00000000
1	8A9BAA0F
2	77B01AAA
3	6GFFAB00
4	ABCDABCD

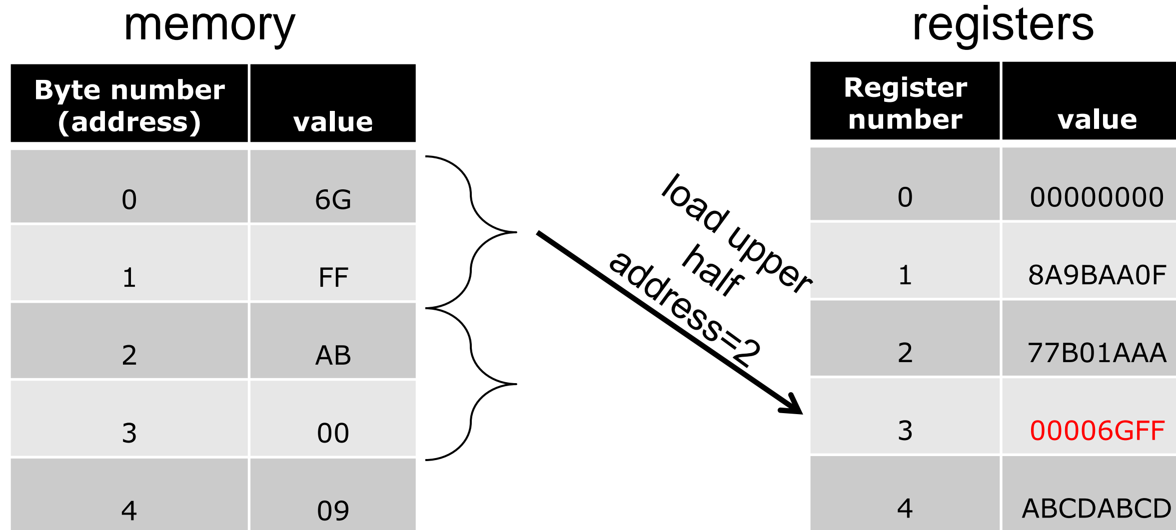
Alignment

- Answer:
  - ALIGNMENT ERROR!!!
  - Segmentation fault



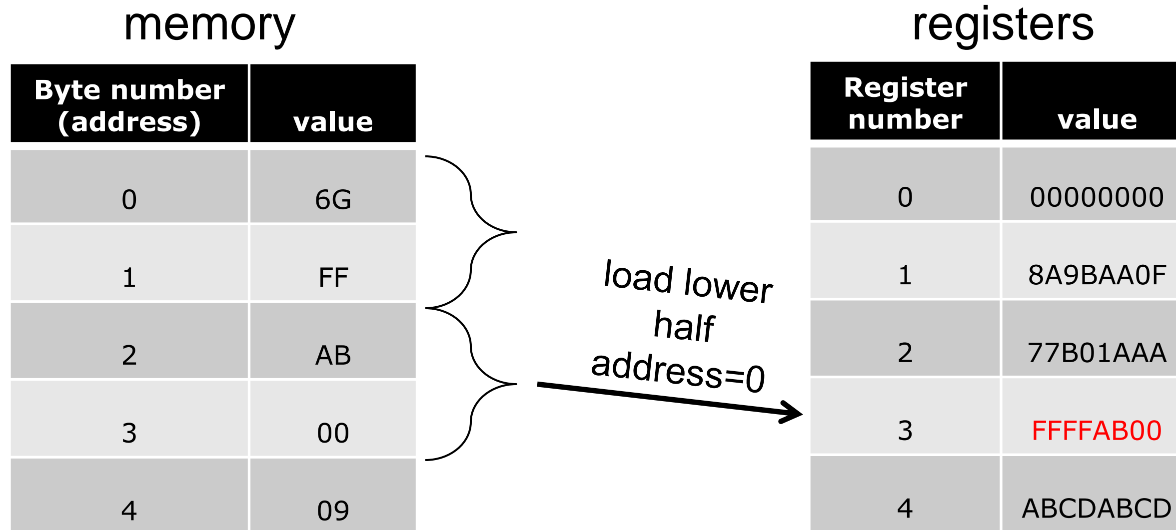
Alignment

- What if I want 16 bits (halfword) from the first word?
  - Want MOST SIGNIFICANT 2 bytes? Use address (offset) 2
  - Want LEAST SIGNIFICANT 2 bytes? Use address (offset) 0



Alignment

- What if I want 16 bits (halfword) from the first word?
  - Want MOST SIGNIFICANT 2 bytes? Use offset 2
  - Want LEAST SIGNIFICANT 2 bytes? Use offset 0

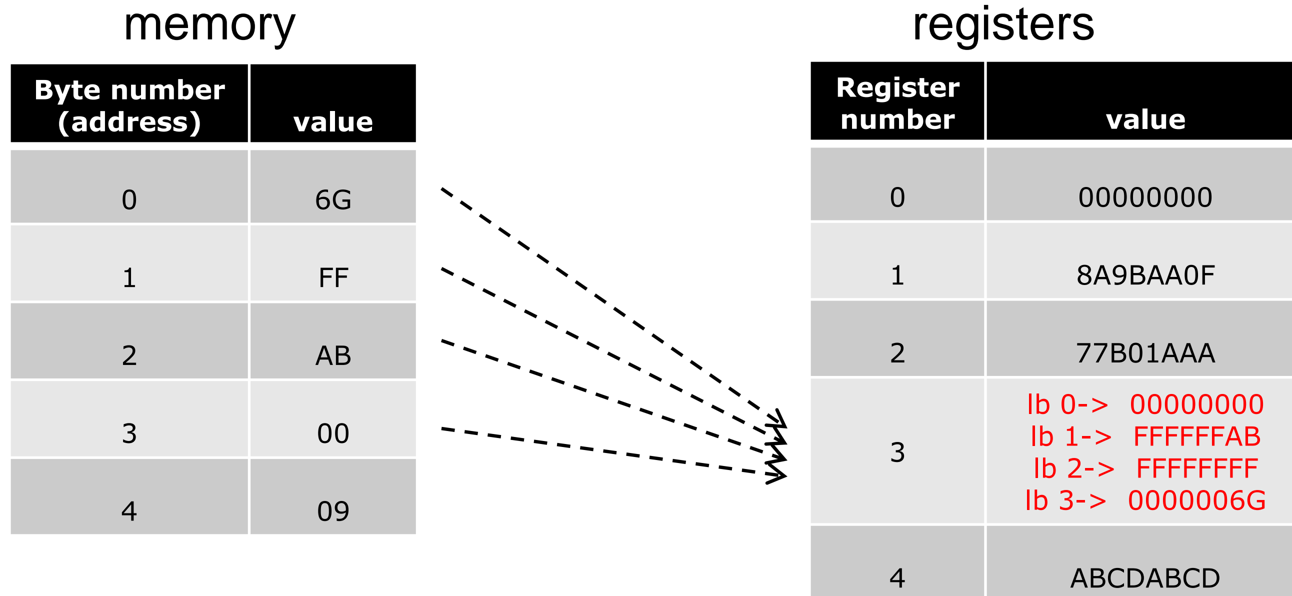


Byte number (address)	value
0	6G
1	FF
2	AB
3	00
4	09

Register number	value
0	00000000
1	8A9BAA0F
2	77B01AAA
3	FFFFAB00
4	ABCDABCD

Alignment

- Load byte?



Load Alignment and Extension

- MEM[0]=DEADBEEF

- LW $1,0($0) -> ($1) <= DEADBEEF
- LW $1,2($0) -> (alignment error)

- LH $1,0($0) -> ($1) <= FFFFBEEF
- LH $1,2($0) -> ($1) <= FFFFDEAD
- LH $1,3($0) -> (alignment error)
- LHU $1,0($0) -> ($1) <= 0000BEEF
- LHU $1,2($0) -> ($1) <= 0000DEAD

- LB $1,1($0) -> ($1) <= FFFFFFBE
- LB $1,3($0) -> ($1) <= FFFFFFDE
- LBU $1,1($0) -> ($1) <= 000000BE
- LBU $1,3($0) -> ($1) <= 000000DE

Store Alignment

- RegFile[$1]=DEADBEEF
- Mem[0]=00000000

- SW $1,0($0) -> Mem[0] <= DEADBEEF
- SW $1,2($0) -> (alignment error)

- SH $1,0($0) -> ($1) <= 0000BEEF
- SH $1,2($0) -> ($1) <= BEEF0000

- SB $1,1($0) -> ($1) <= 0000EF00
- SB $1,3($0) -> ($1) <= EF000000

Hints: Control Unit Design

- Inputs to control unit:
  - Instruction op code, bits 31:26
  - Instruction rt field, bits 20:16 (for single register branches)
  - Instruction function code, bits 5:0 (R-type)
  - Branch outcome (ALU result bit 0, ALU zero output)
  - Stall

- Outputs from control unit:
  - WB signals (destination register, contents)
  - PC source (PC+4, branch target, register target, J-type target)
  - Memory interface signals (read, write)
  - ALU inputs and ALUOp

- Fixed signals:
  - Memory data out (always from B register—contents of rt)
  - Memory address out (always ALU output)