Week 8 Review

Week 8

- Intro to Assembly
- ALU instructions
- Branches and jumps
- Loops

Assembly vs Machine Code

- Machine code is hard to read and write.
- Represent instructions as user-readable code words.
  - User-friendly machine code: one line ←→ one instruction
  - Each processor architecture has its own version.
- Example: `C = A + B`
  - Assume `A` is stored in `$t1`, `B` in `$t2`, `C` in `$t3`.
  - Assembly language instruction:

```
add $t3, $t1, $t2
```

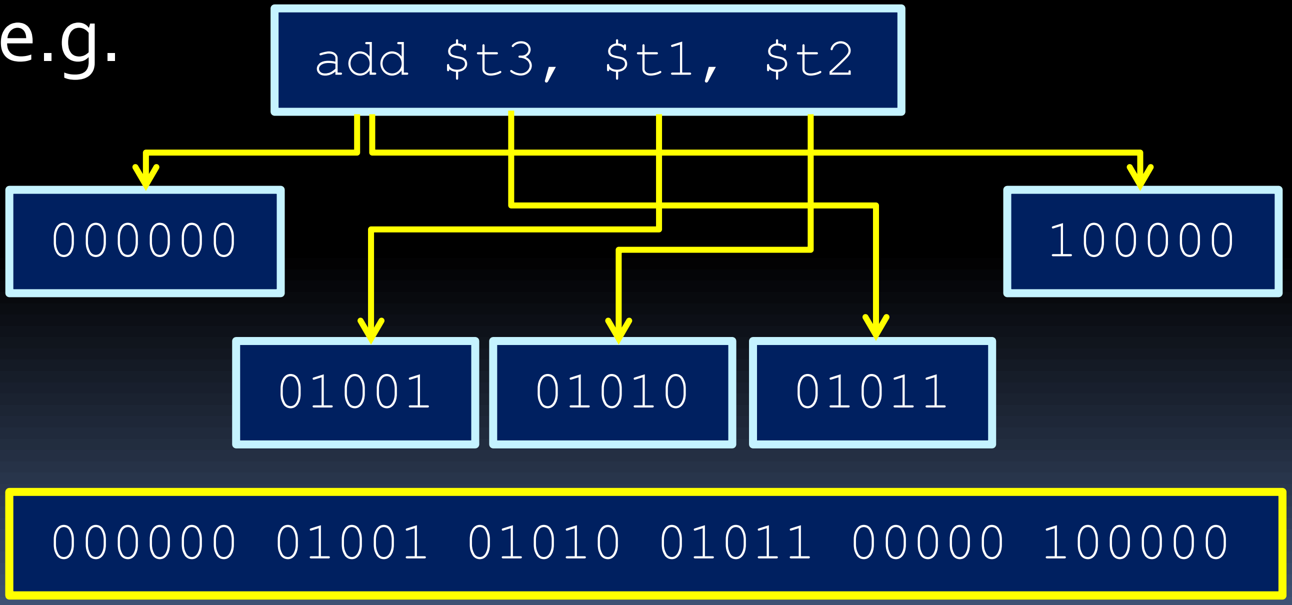
  - Machine code instruction:

```
000000 01001 01010 01011 00000 100000
```

1-to-1 mapping for all assembly code and machine code instructions!

Assembly to Machine Code

- Encoding is reverse of decoding.
- We need to know how to encode the operation to perform, and the register values to operate on.
- e.g.



Warmup

- What are the following assembly language instructions doing?

`sub $t7, $t0, $t1`	Subtract register `$t1` from `$t0` and placing the result into `$t7`
`andi $t7, $t0, 15`	Bitwise AND between register `$t0` and 15 (`1111`), with the result placed into register `$t7`
`sra $t2, $t1, 2`	Arithmetic shift of register `$t1` two bits to the right, with the result stored in `$t2`

What is the instruction type of sra? R-type!

MIPS Register File Registers

Number	Name	Use
0	`$0, $zero`	Always the constant zero
1	`$at`	reserved for assembler (pseudo instructions)
2 – 3	`$v0 - $v1`	function return values
4 – 7	`$a0 - $a3`	function arguments
8 – 15	`$t0 - $t7`	temporary variables
16 – 23	`$s0 - $s7`	saved temporaries
24 – 25	`$t8 - $t9`	temporary variables
26 – 27	`$k0 - $k1`	reserved for operating system kernel
28	`$gp`	global pointer to data segment
29	`$sp`	stack pointer to top of stack
30	`$fp`	frame pointer to function frame start
31	`$ra`	return address from function

Types of Asm Instructions

Arithmetic	`add, mult, …`
Logical	`and, or, …`
Bit shifting	`sll, sra, …`
Data movement	`mflo,mfhi, …`
Branch	`beq, bgtz, …`
Jump	`j, jr, …`
Comparison	`slt, sltu, …`
Memory	`lw, sw, …`

Arithmetic instructions

Instruction	Opcode/Function	Syntax	Operation
add	100000	$d, $s, $t	$d = $s + $t
addu	100001	$d, $s, $t	$d = $s + $t
addi	001000	$t, $s, i	$t = $s + SE(i)
addiu	001001	$t, $s, i	$t = $s + SE(i)
div	011010	$s, $t	lo = $s / $t; hi = $s % $t
divu	011011	$s, $t	lo = $s / $t; hi = $s % $t
mult	011000	$s, $t	hi:lo = $s * $t
multu	011001	$s, $t	hi:lo = $s * $t
sub	100010	$d, $s, $t	$d = $s - $t
subu	100011	$d, $s, $t	$d = $s - $t

Notes: **"hi" and "lo" refer to the HI and LO registers**
"SE" = "sign extend".

Question #1

- Write a piece of assembly code to swap the values in `$t0` and `$t1`, using `$t2` as a temp value.

```
add $t2, $zero, $t0
add $t0, $zero, $t1
add $t1, $zero, $t2
```

Logical instructions

Instruction	Opcode/Function	Syntax	Operation
`and`	100100	\$d, \$s, \$t	\$d = \$s & \$t
`andi`	001100	\$t, \$s, i	\$t = \$s & ZE(i)
`nor`	100111	\$d, \$s, \$t	\$d = ~(\$s | \$t)
`or`	100101	\$d, \$s, \$t	\$d = \$s | \$t
`ori`	001101	\$t, \$s, i	\$t = \$s | ZE(i)
`xor`	100110	\$d, \$s, \$t	\$d = \$s ^ \$t
`xori`	001110	\$t, \$s, i	\$t = \$s ^ ZE(i)

Note: ZE = zero extend (pad upper bits with 0 value).

Shift instructions

Instruction	Opcode/Function	Syntax	Operation
sll	000000	$d, $t, a	$d = $t << a
sllv	000100	$d, $t, $s	$d = $t << $s
sra	000011	$d, $t, a	$d = $t >> a
srav	000111	$d, $t, $s	$d = $t >> $s
srl	000010	$d, $t, a	$d = $t >>> a
srlv	000110	$d, $t, $s	$d = $t >>> $s

- Order is $d, $t, $s or $d, $t, a (not $d, $s, $t as before!)
- `srl` = "shift right logical"
- `sra` = "shift right arithmetic".
- The "`v`" denotes a variable number of bits, specified by `$s`.
- `a` is shift amount, and is stored in `shamt` when encoding the R-type machine code instructions.

lui – load upper immediate

Instruction	Opcode/Function	Syntax	Operation
`lui`	001111	`$t, i`	`$t = i << 16`

- Load 16-bit immediate into upper half of the register.
- The lower 16 bits of the register are set to zero.

```
iiiiiiiiiiiiiiii0000000000000000
```

li pseudoinstruction

Instruction	Opcode/Function	Syntax	Operation
li	N/A	$t, i	$t ← i

- Load immediate into register.
- If immediate fits in 16-bit, uses `addiu`
- If immediate is 32-bit, uses `lui` followed by `ori`

```
li $s0,0x1234ABCD
```

```
lui $s0,0x1234
ori $s0,$s0,0xABCD
```

Formatting Assembly Code

- Start file with `.text`
  - (we'll see other options later)
- Follow this with:
- `.globl main`
  - (Makes the main label visible to the OS)
- `main:`
  - (Tells OS which line of code should run first.)
- Write instructions
  - `label:  <instr> <params>    # comments`
  - Labels and comments as needed
- Use `#` for comments. Comments are critical!
- At the end of the program, tell the OS to finish:

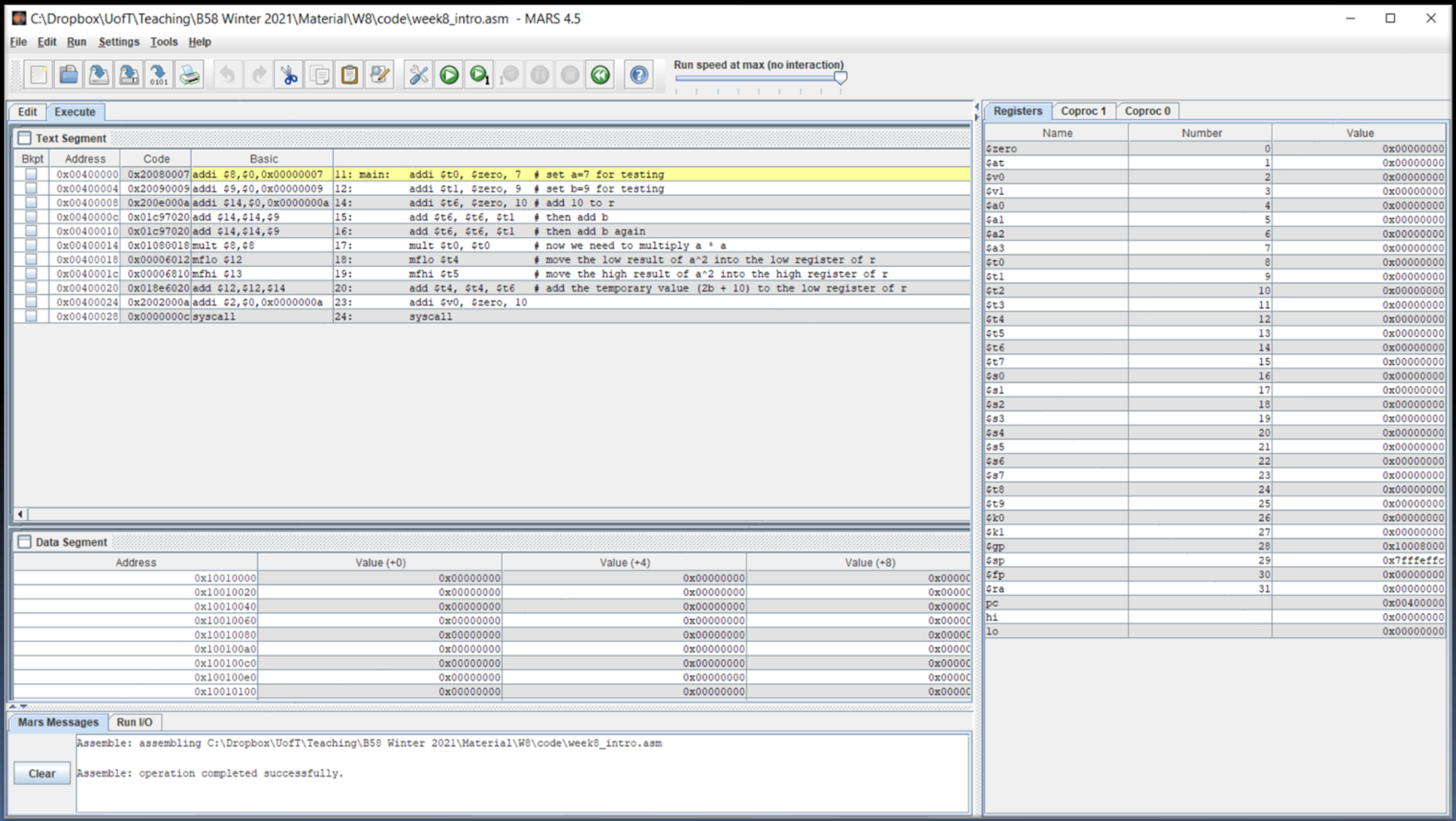
```
li $v0, 10
syscall
```

```
.text

.globl main
main:
        <code>

        li $v0, 10
        syscall
```

```
# Compute the following result: r = a^2 + 2b + 10
.text

.globl main
# $t0 will be a,   $t1 will be b,   $t5 will be r
# $t6 will be temp
main:
      addi $t0, $zero, 7   # set a=7 for testing
      addi $t1, $zero, 9   # set b=9 for testing

      addi $t6, $zero, 10  # add 10 to r
      add $t6, $t6, $t1    # then add b
      add $t6, $t6, $t1    # then add b again
      mult $t0, $t0        # multiply a * a
      mflo $t4             # move the low result of a^2
                           # into the register for r
      add $t4, $t4, $t6    # add the temporary value
                           # (2b + 10) to the result

      addi $v0, $zero, 10  # end program
      syscall
```

MARS Simulator

- Use it to write and run MIPS assembly programs.



Control Flow in Assembly

- Assembly is not sophisticated.
  - We have to tell it manually where to go and when.
- Labels indicate points in the program we might need to jump to.
- Branch instructions tell the CPU to go somewhere based on some condition.

  `beq $t0, $t5, label` → if $to = $t5, jump to `label`
- We can also jump unconditionally.

  `j label` → jump to `label` (always)

Warmup

- What are the following assembly language instructions doing?

`bgtz $t2, TOP`

Jump to the line with label "TOP" if register `$t2` is greater than `0` (`$zero`)
Comparison is signed!

`jalr $t0`

Store the current PC location into `$ra` (register `$31`) and jump to the location whose address is stored in register `$t0`

Practice, and learn the meaning behind the names of instructions

Branch instructions

Instruction	Opcode/Function	Syntax	Operation
`beq`	000100	\$s, \$t, label	if (\$s == \$t) pc ← label
`bgtz`	000111	\$s, label	if (\$s > 0) pc ← label
`blez`	000110	\$s, label	if (\$s ≤ 0) pc ← label
`bne`	000101	\$s, \$t, label	if (\$s != \$t) pc ← label

- These comparisons are signed.
- Branch operations are key to implementing if/else statements and loops.
- The labels are memory locations, assigned to each label at compile time.

Comparison instructions

Instruction	Opcode/Function	Syntax	Operation
slt	101010	$d, $s, $t	$d = ($s < $t)
sltu	101001	$d, $s, $t	$d = ($s < $t)
slti	001010	$t, $s, i	$t = ($s < SE(i))
sltiu	001001	$t, $s, i	$t = ($s < SE(i))

- "slt" = "Set Less Than"
- Comparison operation stores one (1) in the destination register if the less-than comparison is true, and stores a zero in that location otherwise.
- Signed: `0x8000000` is less than all other numbers
- Unsigned: `0 - 0x7FFFFFFF` are less than `0x8000000`
  - Immediate is sign-extended even in `sltiu`

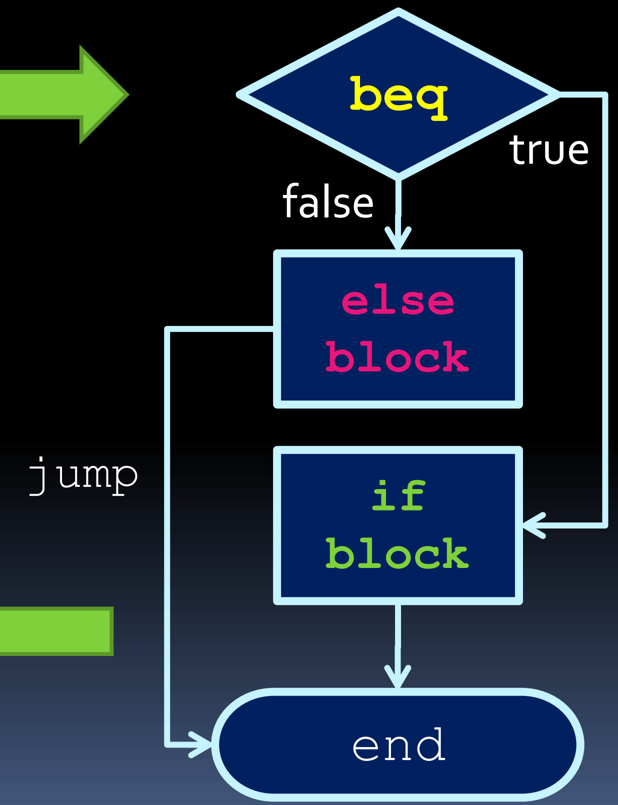
Branch Pseudoinstructions

- Implemented using `slt` variants and branches.
- You are allowed to use them unless we say otherwise.

Instruction	Opcode/Function	Syntax	Operation
blt	N/A	\$s, \$t, label	if (\$s < \$t) pc ← label
bltu	N/A	\$s, \$t, label	if (\$s < \$t) pc ← label
bgt	N/A	\$s, \$t, label	if (\$s > \$t) pc ← label
bgtu	N/A	\$s, \$t, label	if (\$s > \$t) pc ← label
ble	N/A	\$s, \$t, label	if (\$s ≤ \$t) pc ← label
bleu	N/A	\$s, \$t, label	if (\$s ≤ \$t) pc ← label
bge	N/A	\$s, \$t, label	if (\$s ≥ \$t) pc ← label
bgeu	N/A	\$s, \$t, label	if (\$s ≥ \$t) pc ← label

If statements

```
if ( i == j )
  i = i+1;
else
  j = j-1;
j = j+i;
```




```
#  $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF
         addi $t2, $t2, -1
         j END
IF:      addi $t1, $t1, 1
END:     add $t2, $t2, $t1
```

If/Else using beq

```
if ( i == j )
   i = i+1;
else
   j = j-1;
j = j+i;
```

```
#  $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF     # branch if ( i == j )
         addi $t2, $t2, -1     # j--
         j END                 # jump over IF
IF:      addi $t1, $t1, 1      # i++
END:     add $t2, $t2, $t1     # j += i
```

If/Else using bne

```
if ( i == j )
   i = i+1;
else
   j = j-1;
j = j+i;
```

```
#  $t1 = i, $t2 = j
main:    bne  $t1, $t2, ELSE   # branch if ( i != j )
         addi $t1, $t1, 1      # i++
         j END                 # jump over ELSE
ELSE:    addi $t2, $t2, -1     # j--
END:     add $t2, $t2, $t1     # j += i
```

Multiple Conditions Inside If

```
if ( i == j && i == k )
   i++ ;  // if-body
else
   j-- ;  // else-body
j = i + k ;
```

- Multiple branches whose flow matches if logic.

```
#  $t1 = i, $t2 = j, $t3 = k
main:  bne $t1, $t2, ELSE   # cond1: branch if ( i != j )
       bne $t1, $t3, ELSE   # cond2: branch if ( i != k )
IF:    addi $t1, $t1, 1     # if (i==j|i==k) → i++
       j END                # jump over else
ELSE:  addi $t2, $t2, -1    # else-body: j--
END:   add $t2, $t1, $t3    # j = i + k
```

Week 9: Memory and functions

Last Week

- Assembly basics
- ALU operations
  - Arithmetic
  - Logical
  - Shift
- Branches (conditions and loops)
- Pseudoinstructions

```
addi $t6, $zero, 10
add $t6, $t6, $t1
add $t6, $t6, $t1
mult $t0, $t0
mflo $t4
add $t4, $t4, $t6
```

```
main:  add $t0, $0, $0
       addi $t1, $0, 100
LOOP : beq $t0, $t1, END
       addi $t0, $t0, 1
       j LOOP
END:
```

Assembly code example

- Fibonacci sequence in assembly code:

```
# fib.asm
# register usage: $t3=n, $t4=f1, $t5=f2
#
FIB:  addi $t3, $zero, 10     # initialize n=10
      addi $t4, $zero, 1      # initialize f1=1
      addi $t5, $zero, 1      # initialize f2=-1
LOOP: beq $t3, $zero, END     # done loop if n==0
      add $t4, $t4, $t5       # f1 = f1 + f2
      sub $t5, $t4, $t5       # f2 = f1 - f2
      addi $t3, $t3, -1       # n = n - 1
      j LOOP                  # repeat until done
END:                          # result in f1 = $4
```

Making sense of assembly code

- Assembly language looks intimidating because the programs involve a lot of code.
  - No worse than your CSCA08 assignments would look to the untrained eye!
- The key to reading and designing assembly code is recognizing portions of code that represent higher-level structures that you're familiar with.
  - Operators, loops, if/else, function calls, etc.

OS Services

- Sometimes we want to invoke OS services.
  - Input/output, files, get time, ending the program…
- These are known as system calls
- Mechanism:
  1. Set `$v0` to the number of the service.
  2. Use `$a0-$a3` to pass arguments to the service.
  3. Invoke the `syscall` instruction
  4. Read results, if any, from registers (usually `$a0`)
- We'll learn more about this in a future week.

Some MARS Services

Service	Code in $v0	Input/Output
print_int	1	$a0 is int to print
print_string	4	$a0 is address of ASCIIZ string to print
read_int	5	$v0 is int read
read_string	8	$a0 is address of buffer $a1 is buffer size in bytes
exit	10	
open_file	13	$a0 is address of ASCIIZ string containing file name $a1 is flag $a2 is mode $v0 is file descriptor
read_from_file	14	$a0 is file descriptor $a1 is address of input buffer $a2 is number of characters to read
write_to_file	15	$a0 is file descriptor $a1 is address of output buffer $a2 is number of characters to write
close_file	16	$a0 is file descriptor

syscall example

```
.text
.globl main
main:
        # Read a number (result will be in $v0)
        li $v0, 5
        syscall

        addi $t0, $v0, 1     # $t0 = $v0 + 1

        # Print result
        li $v0, 1
        move $a0, $t0        # it will print the number in $a0
        syscall

        # End program
        li $v0, 10
        syscall
```

Interacting With Memory

Interacting with memory

- All of the previous instructions perform operations on registers and immediate values.
  - What about memory?
- All programs must load values from memory into registers, operate on them, and then store the values back into memory.
- Memory operations are I-type, with the form:

```
lw  $t0, 12($s0)
```

Load or store

Register storing address of data value in memory

Local data register

Offset from memory address

Load/Store in Datapath

- The terms “load” and “store” are seen from the perspective of the processor, looking at memory.
- Again, memory operations are I-type.
- Loads are memory read operations.
  - We load (i.e., read) from memory.
  - We load a value **from** a memory address into a register.
- Stores are memory write operations.
  - We store (i.e., write) a data value from a register **to** a memory address.
  - Store instructions do not have a destination register, and therefore do not write to the register file.

Memory Instructions in MIPS assembly

When loading a byte or a half-word you can choose u for unsigned. Leave it blank as for all other cases.

Specifies the location to access as MEM[$s + SE(i)]

```
? ? ?    $t, i($s)
```

l for load or
s for store

b for byte,
h for half-word,
w for word

Destination register for loads, source register for stores.

Load & store instructions

Instruction	Opcode/Function	Syntax	Operation
`lb`	100000	$t, i ($s)	$t = SE (MEM [$s + i]:1)
`lbu`	100100	$t, i ($s)	$t = ZE (MEM [$s + i]:1)
`lh`	100001	$t, i ($s)	$t = SE (MEM [$s + i]:2)
`lhu`	100101	$t, i ($s)	$t = ZE (MEM [$s + i]:2)
`lw`	100011	$t, i ($s)	$t = MEM [$s + i]:4
`sb`	101000	$t, i ($s)	MEM [$s + i]:1 = LB ($t)
`sh`	101001	$t, i ($s)	MEM [$s + i]:2 = LH ($t)
`sw`	101011	$t, i ($s)	MEM [$s + i]:4 = $t

- “b”, “h” and “w” correspond to “byte”, “half word” and “word”, indicating the length of the data.
- “SE” stands for “sign extend”, “ZE” stands for “zero extend”.

Examples

```
lh  $t0, 12($s0)
```

Load a half-word (2 bytes) starting from `MEM[$s0 + 12]`, sign-extend it to 4 bytes and store in register `$t0`

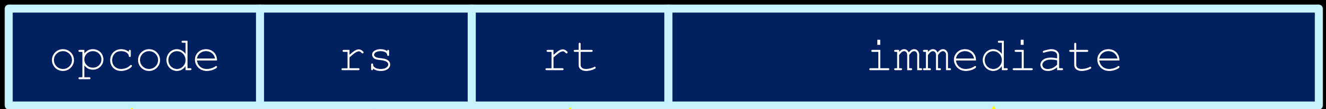
```
sb  $t0, 12($s0)
```

Take the lowest byte of the word stored in register `$t0`, store it to memory starting from address `$s0 + 12`

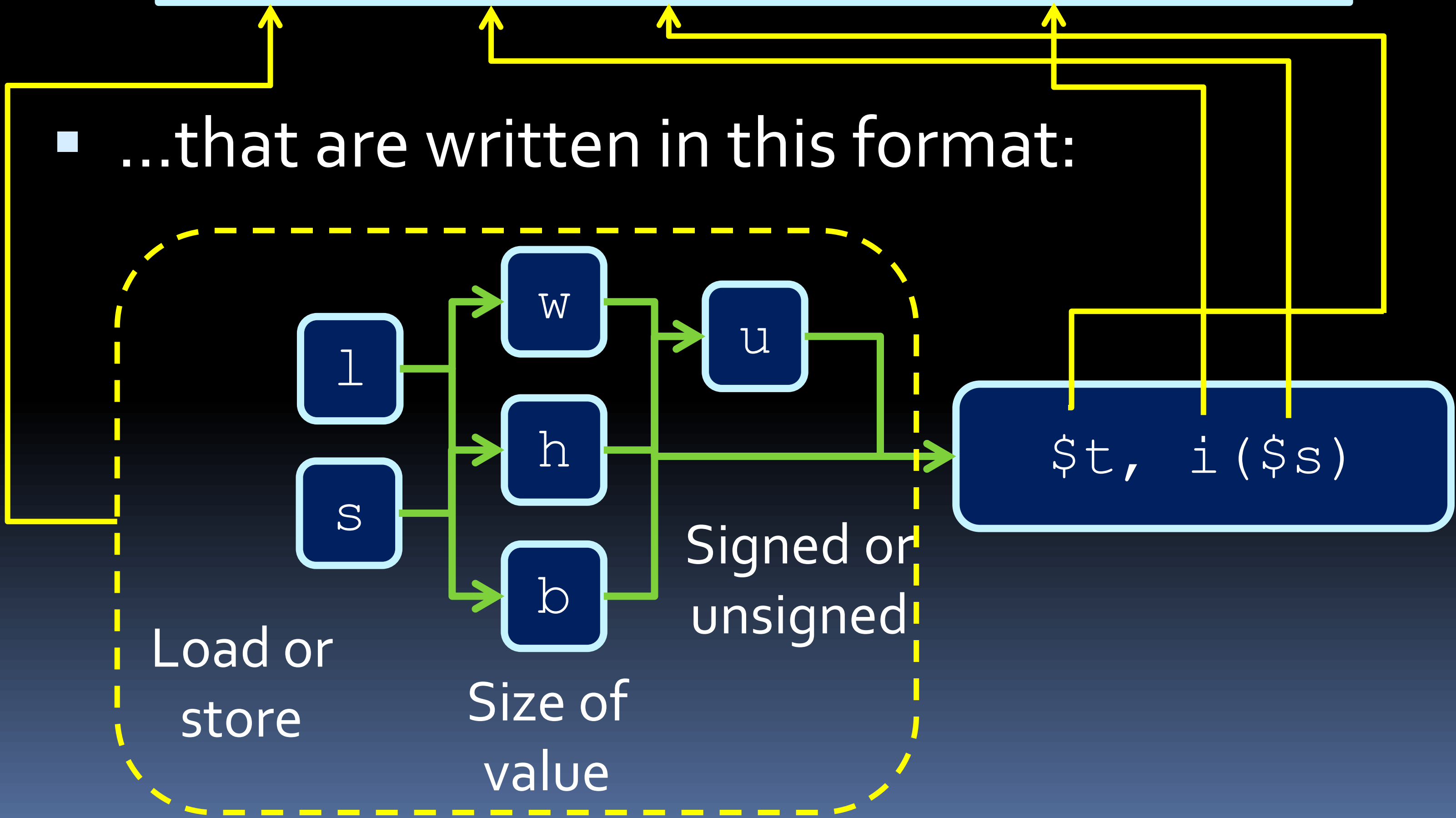
As we'll see soon, base + offset (`$s0 + 12`) is very useful for dealing with structs, stack, arrays and more.

Machine Code for Load/Store

- Load & store instructions are I-type operations:

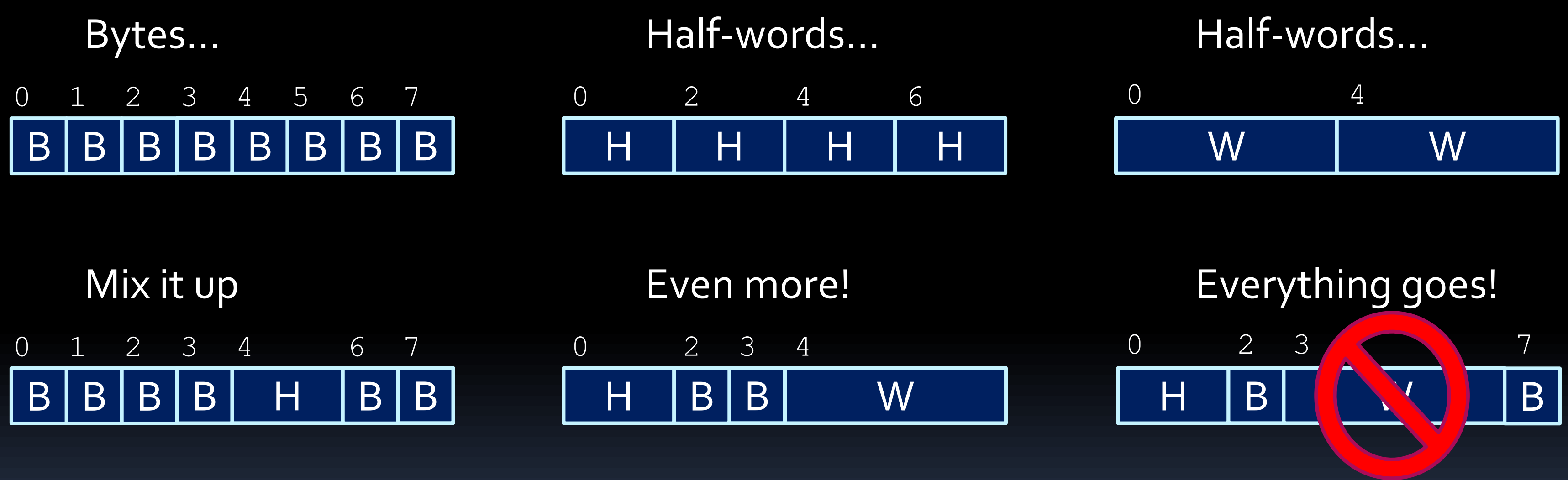


- …that are written in this format:



A View of Memory

- The memory doesn't know what it is storing.
- Same memory can be viewed as…



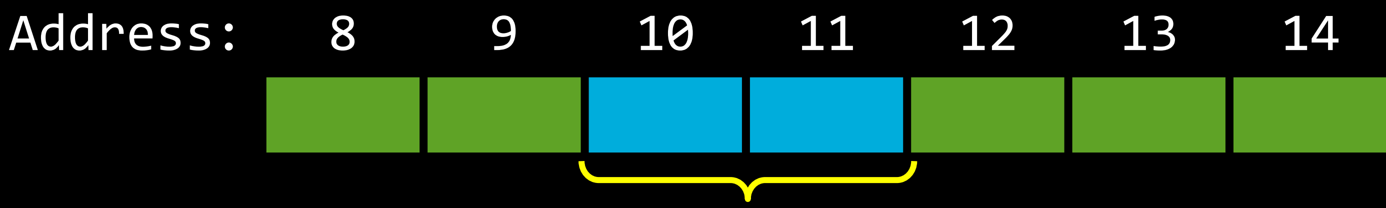
The last one will crash your program.

Alignment Requirements

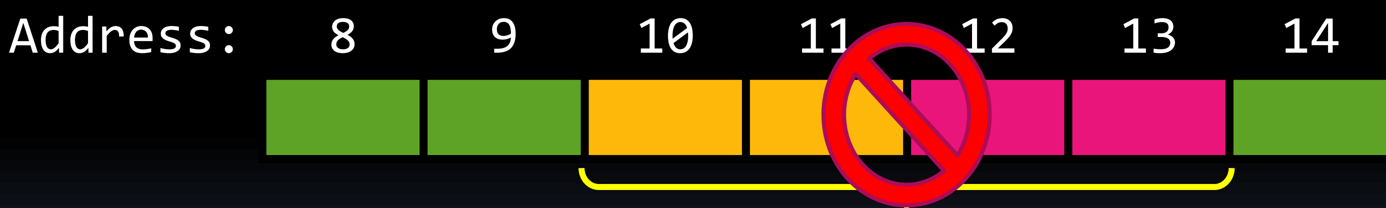
- Misaligned memory accesses result in errors.
  - Causes an exception (more on that, later)
- Word accesses (i.e., addresses specified in a `lw` or `sw` instruction) should be word-aligned (divisible by 4).
- Half-word accesses should only involve half-word aligned addresses (i.e., even addresses).
- No constraints for byte accesses.

Alignment Examples

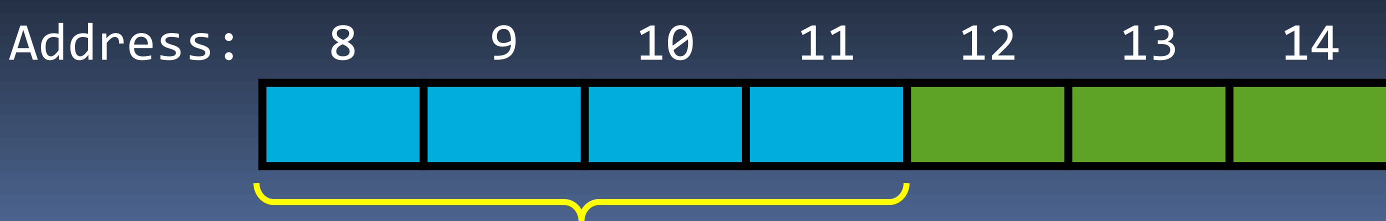
- Access to half-word at address 10 is aligned



- Access to word at address 10 is unaligned

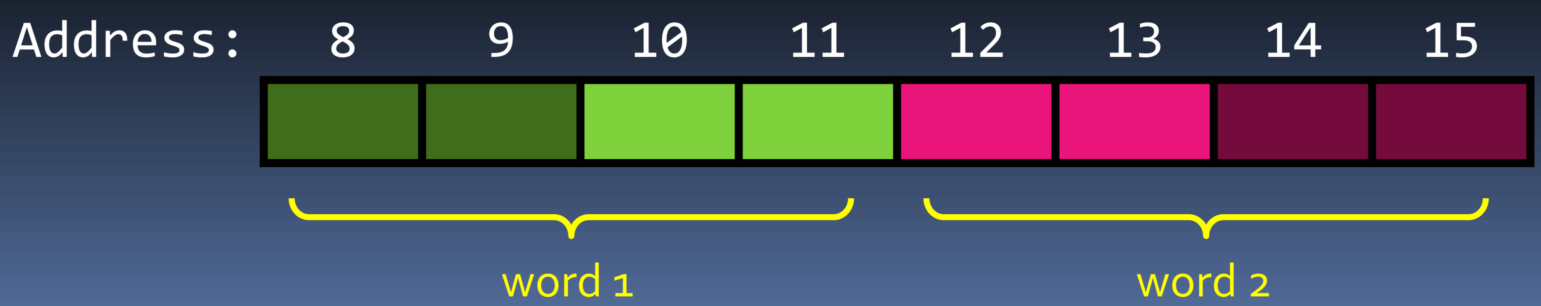


- Access to word at address 8 is aligned



Why Alignment Limitations?

- MIPS memory model is byte-addressable
  - Each byte has an address.
- In reality, memory is arranged in 32-bit units
  - RAM is arranged in words, 32 data lines connecting to each row.
- Loading unaligned data would mean fetching two words from memory and combining them:



Little Endian vs. Big Endian

- Let's say we want to store the word `0x1234ABCD` starting from address X.
- How do we split the multiple bytes across memory X to X+3?
- Same in reading: if reading word (4 bytes) from address X, how do we assemble the bytes?

Address	Byte
X	??
X + 1	??
X + 2	??
X + 3	??

Big Endian vs. Little Endian

- Big Endian ("big end first")
  - The most significant byte of the word is stored first (i.e., at address X). The 2nd most significant byte at address X+1 and so on.

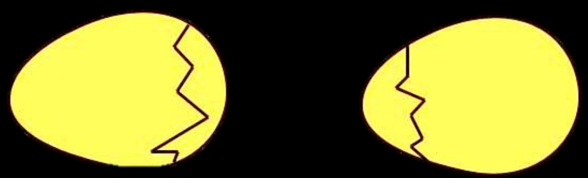
Address	Byte
X	12
X + 1	34
X + 2	AB
X + 3	CD

- Little Endian ("little end first")
  - The least significant byte of the word is stored first (i.e., at address X). The 2nd least significant byte at address X+1 and so on.

Address	Byte
X	CD
X + 1	AB
X + 2	34
X + 3	12

Endianness Headaches

- Examples
  - x86 CPUs are little-endian.
  - AArch64 CPUs are big-endian.
  - 32-bit ARM is bi-endian: can built/configured to be either.



- Causes headaches when transferring data between computers.
  - Good protocols specify endianness of the data representation.
- MIPS is bi-endian.
- Our MARS simulator is little-endian.

More Pseudo-instructions

Instruction	Opcode/Function	Syntax	Operation
la	N/A	$t, label	$t = address(MEM [label])
move	N/A	$t,$s	$t = $s
li	N/A	$t, i	$t = i

- Remember: these aren't really MIPS instructions
  - Simplifications of multiple instructions.
  - But they make life way easier.
- We already saw `li` (`lui` followed by `ori`)
- `la` loads the address of a label.
  - Implemented similarly to `li`

Labeling Data Storage

- Also known as variables
- At beginning of program, create labels for memory locations that are used to store values.
- Always in form: `label:   .type   value(s)`

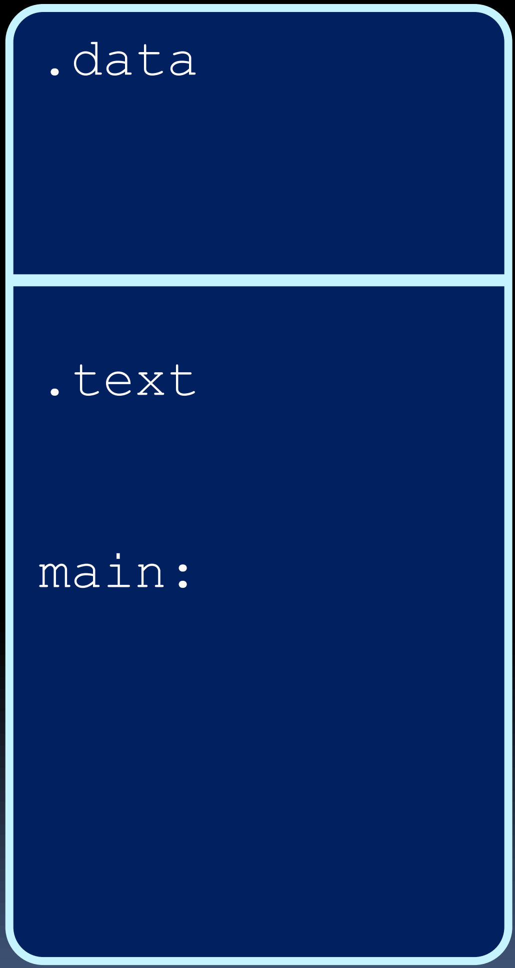
```
# create a single integer variable with initial value 3
var1:        .word    3

# create a 4-element array of integers
var1:        .word    2, -4, 100, 2

# create a 3-element character (byte) array with
# elements initialized to a and b
array1:      .byte    'a','b'

# allocate 40 consecutive bytes, with uninitialized
# storage. Could be used as a 40-element character
# array, or a 10-element integer array.
array2:      .space   40
```

Remember the C Datatypes

- int is word
- short is half
- char is byte

Memory Sections & syntax

- Programs are divided into two main sections in memory:
- `.data` indicates the start of the data values section.
  - Typically placed at beginning of program.
  - Labels inside .data section indicate variables.
- `.text` indicates the start of the instructions section.
  - Labels inside the .text sections are program labels and branch addresses.
  - The label `main:` indicates the initial line of code to run when executing the program.
  - Other labels are for functions, branches, etc.



Example

- Given variables
  - RES – integer (32-bit)
  - A – integer
  - B – integer
  - C – half-word (16-bit)
- Implement RES = A*B+C
  - Compute A*B+C…
  - …and store the result in RES

```
.data
RES:    .word    0
A:      .word    5
B:      .word   -2
C:      .half    -7
```

RES = A*B+C

- Assign registers
  - A in $t0, B in $t1, C in $t2
  - Store temporary result in $t9
  - Use $t5 to store addresses of variables.
- Implement:
  - Load address of each variable
  - Load value from the address into register
  - Compute into $t9
  - Load address of result variable
  - Store result from $t9 into memory

```
.data
RES:    .word    0
A:      .word    5
B:      .word   -2
C:      .half   -7
```

RES = A*B+C

```
.data
RES:    .word     0
A:      .word     5
B:      .word    -2
C:      .half    -7

.text
.globl main
main:      la $t5, A          # get address of A
           lw $t0, 0($t5)     # load value of A
           la $t1, B          # get address of B
           lw $t1, 0($t1)     # load value of B
           la $t2, C          # get address of C
           lh $t2, 0($t2)     # load value of C

           mult $t0, $t1      # compute A*B
           mflo $t9
           add $t9,$t9,$t2 # add C

           la $t5, RES        # get address of RES
           sw $t9, 0($t5)     # store result in RES
```