Wrapping Up the CO Course

Some highlights in preparation for the final exam

Matrix operations in MIPS

A[4] A_5
A[3] A_4
A[2] A_3
A[1] A_2
A[0] A_1

B_{53}
B_{52}
B_{51}
B_{43}
B_{42}
B_{41}
B_{33}
B_{32}
B_{31}
B_{23}
B_{22}
B_{21}
B_{13}
B_{12}
B_{11}

$$C = [A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5] \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & \\ B_{31} & B_{32} & \\ B_{41} & B_{42} & \\ B_{51} & B_{52} & B_{53} \end{bmatrix}$$

$$C = [C_1 \quad C_2 \quad C_3]$$

$$C_1 = A_1 B_{11} + A_2 B_{21} + \cdots + A_5 B_{51}$$

$$C_j = \sum_{i=1}^{5} A_i B_{ij}$$

HW with ASCII: Review

- **Loop:**# loop through A, check index bounds
- slt $t0, $s2, $s4 # if (i < g), $t0 = 1
- beq $t0, $zero, end
- lw $a0, 0($s6) # load A[i]
- jal fact # now call fact
- # convert v0 to an ascii character
- addi $v0, $v0, 48
- #store character into msgresults[2i]
- sb $v0, 0($s5)
- addi $s2, $s2, 1 # i++ update the index i
- # update the addresses
- addi $s6, $s6, 4 # addr(A[i]) += 4
- addi $s5, $s5, 2 # msgresults[2i]: this is a byte array
- **j Loop** # loop again
- end:

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Note about mult and div

- ***mult & multu*** are native MIPS instructions
- The result is 64-bit value
- ***mul*** is pseudoinstruction: it makes it look as if MIPS has a 32-bit multiply instruction that places its 32-bit result:
  - mul d,s,t # multiply $s by $t. put the result in $d
- There is no overflow checking. The bits of hi are not examined nor saved.
- There is a similar pseudoinstruction for division.
- The native ***div s,t*** and ***divu s,t*** put their results in the MIPS registers Hi and Lo; the 32-bit quotient goes in lo and the 32-bit remainder goes in hi.
- To move the quotient into a register, ***mflo*** is used; for the remainder, use ***mfhi (script divshow)***

Remainder – Even or Odd?

0	1	0	0	1	0	0	1

$X=0\times2^7 + 1\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$
$=0 + 64 + 0 + 0 + 8 + 0 + 0 + 1 = \mathbf{73}$

73/2=36   73%2=1

srl Y, X, 1

0	0	1	0	0	1	0	0

$Y=0\times2^7 + 0\times2^6 + 1\times2^5 + 0\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0$
$=0 + 0 + 32 + 0 + 0 + 4 + 0 + 0 = \mathbf{36}$

sll Z, Y, 1

0	1	0	0	1	0	0	0

$Z=0\times2^7 + 1\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0$
$=0 + 64 + 0 + 0 + 8 + 0 + 0 + 0 = \mathbf{72}$

X-Z=1

Remainder – Divisible by 4?

0	1	0	0	1	0	0	1

$X=0\times2^7 + 1\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$
$=0 + 64 + 0 + 0 + 8 + 0 + 0 + 1 =$ **73**

73/2=36 73%2=1

srl Y, X, 2

0	0	0	1	0	0	1	0

$Y=0\times2^7 + 0\times2^6 + 0\times2^5 + 1\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$
$=0 + 0 + 0 + 16 + 0 + 0 + 2 + 0 =$ **18**

sll Z, Y, 2

0	1	0	0	1	0	0	0

$Z=0\times2^7 + 1\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0$
$=0 + 64 + 0 + 0 + 8 + 0 + 0 + 0 =$ **72**

R=X-Z=1

beq R, zero, Divis

Data declarations

- **.space** reserves n bytes of memory, without aligning. e.g. **arr: .space 100**
- .byte stores the n ***values*** in successive bytes of memory. e.g. **num: .byte 0x01, 0x03**
- **.word** store n 32-bit words contiguously in aligned memory. e.g. **val: .word 10, -14, 30**
- **.ascii** stores string in memory **without a null terminator**. e.g. str: .ascii "Hello, world"
- **.asciiz** stores string in memory **with a null terminator**. e.g. **str: .asciiz "Hello, world“** - exactly like .ascii with a .byte ‘\0’ after it.
- **.align** aligns the next data on a 2^n byte boundary. e.g. .align 2 aligns the next value on a word boundary. On the other hand if n is 0 then alignment is turned off until next data segment.

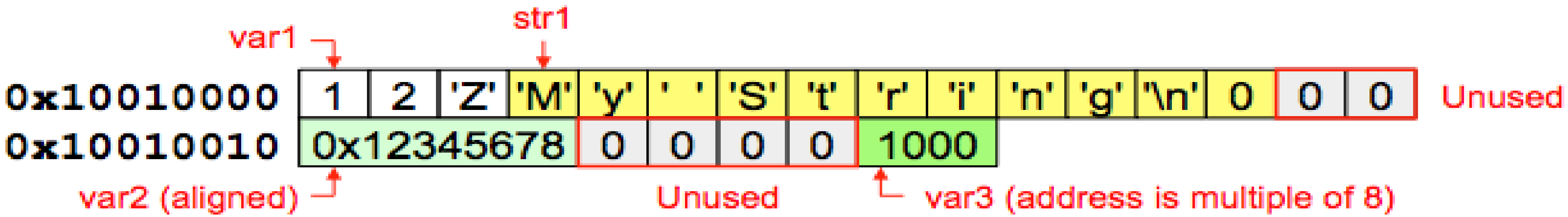
Example of the data memory layout

- Assembler builds a symbol table for labels (variables)
  - Assembler computes the address of each label in data segment
- Example

```
.DATA
var1:    .BYTE    1, 2,'Z'
str1:    .ASCIIZ "My String\n"
var2:    .WORD    0x12345678
.ALIGN   3
var3:    .HALF    1000
```

Symbol Table

Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018



Loading a 32-Bit Constant

What is the MIPS assembly code to load this 32-bit constant into register $s0?

- C=0000 0000 0011 1101 0000 1001 0000 0000

We cannot use addi $s0, $zero, C because addi only had 16 bits for the constant

- First, we would load the upper 16 bits, which is 61 in decimal, using lui:
- lui $s0, 61 # 61 decimal = 0000 0000 0011 1101 binary
- The value of register $s0 afterward is
- 0000 0000 0011 1101 0000 0000 0000 0000
- The next step is to insert the lower 16 bits, whose decimal value is 2304:
- ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
- The final value in register $s0 is the desired value:
- 0000 0000 0011 1101 0000 1001 0000 0000

Do NOT use non-native address formats

- theArray: .space 160
-   .text
- main:
-   la $s0, theArray
-   li $t6, 1   # Sets t6 to 1
-   li $t7, 4   # Sets t7 to 4
-   #sw $t6, theArray($0)   # Sets the first term to 1 ←
-   sw $t6, 0($s0)   # Sets the first term to 1
-    #sw $t6, theArray($t7) # Sets the second term to 1 ←
-   add $s1, $s0, $t7
-   sw $t6, 0($s1) # Sets the second term to 1
-   li $t0, 8   # Sets t0 to 8
- *Example: fibarray vs fibarray_correct*

Final: Example questions

- Loop: Load elements from an integer array, check if the loaded value satisfies a given condition (e.g. divisibility), then add to the sum
- Load floating point elements from memory locations described by an indirect address (e.g. A[B[4i]-2i]) and perform an operation(e.g. A[2i] =2 x A[B[4i]-2i])
- Fill in the blanks for a function implementation
  - e.g. Function:

```
----------------------------
addi $s0, $s0, 4
mul  $v0, $a0,$s0
-----------------------------
jr $ra
```

- Interpret a floating point value   (like in the HW)

About combining integers & floating points

- E.g. to sum up an integer in a3 with an floating point in f2

- mtc1 $a3, $f1 #convert integer to fp
- cvt.s.w $f1, $f1
- add.s $f2, $f2, $f1 #then add to the fp

- HW note:
- Depending on whether you rounded **1.2414340730756521**
- **as 1.24143407307 or 1.24143407308 you got an error of 102 or 26**