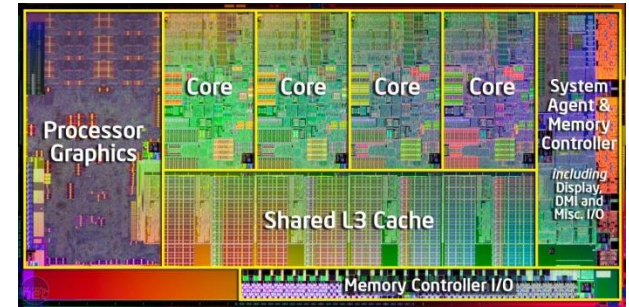




고려대학교
KOREA UNIVERSITY



COSE222, COMP212 Computer Architecture

Lecture 4. Miscellaneous

Addressing Mode, Memory Map, Pointer, and ASCII

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

Addressing Modes

- The addressing mode defines how instructions identify their operands
 - Addressing modes typically refer to diverse ways of specifying the memory address of an operand by using information held in registers and/or constants contained within a machine instruction
 - It is one aspect of ISA in CPU
- Addressing modes in MIPS
 - Register-only
 - Immediate
 - Base Addressing
 - PC-Relative
 - Pseudo Direct

MIPS Addressing Modes

- Register-only Addressing

- R-type instructions use the register-only addressing
- Operands are found in registers

```
add $s0, $t2, $t3
```

```
sub $t8, $s1, $0
```

- Immediate Addressing

- Some I-type instructions use the immediate addressing
- 16-bit immediate is used as an operand

```
addi $s4, $t5, -73
```

```
ori  $t3, $t7, 0xFF
```

MIPS Addressing Modes

- Base Addressing

- The address of the memory is found by adding the base address (r_s) to the sign-extended 16-bit offset (imm)
- Address calculation of an operand

base address + sign-extended immediate

`lw $s4, 72($0) // address = $0 + 72`

`sw $t2, -25($t1) // address = $t1 - 25`

MIPS Addressing Modes

- PC-Relative Addressing

- Conditional branch instructions use PC-relative addressing to specify the new PC if the branch is taken
 - The signed offset in the immediate field is added to the PC to obtain the new PC
 - Thus, the branch destination address is said to be relative to the current PC

0x10	beq	\$t0, \$0, else
0x14	addi	\$v0, \$0, 1
0x18	addi	\$sp, \$sp, i
0x1C	jr	\$ra
0x20	else:	addi \$a0, \$a0, -1
0x24	jal	factorial

Assembly Code

Field Values

	op	rs	rt	imm
beq \$t0, \$0, else	4	8	0	3
(beq \$t0, \$0, 3)	6 bits	5 bits	5 bits	5 bits 5 bits 6 bits

MIPS Addressing Modes

- Pseudo-direct Addressing

- In **direct addressing**, an address is specified in the instruction
- But, the J-type instruction encoding does not have enough bits to specify a full 32-bit jump address
 - Opcode uses the 6 bits of the instructions
 - So, only 26-bits are left to encode the jump target address
- MIPS calculates the jump target address from the J-type instruction by appending two 0's and prepending the four MSBs of PC + 4

$\text{destination} = \{(\text{PC}+4)[31:28] , \text{jump target}, 2'b00\}$

ASCII Encoding

- In old days, exchanging text between computers was difficult because early computers lacked a standard mapping between bytes and characters
- In 1963, the American Standards Association published the **American Standard Code for Information Interchange (ASCII)**
 - ASCII assigns each text character a unique byte value
 - C language uses the type `char` to represent a character

ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Example

```
#include <stdio.h>

int main()
{
    char aa, bb;
    char * addr_aa;
    char * addr_bb;

    aa = 'A';
    bb = 'a';

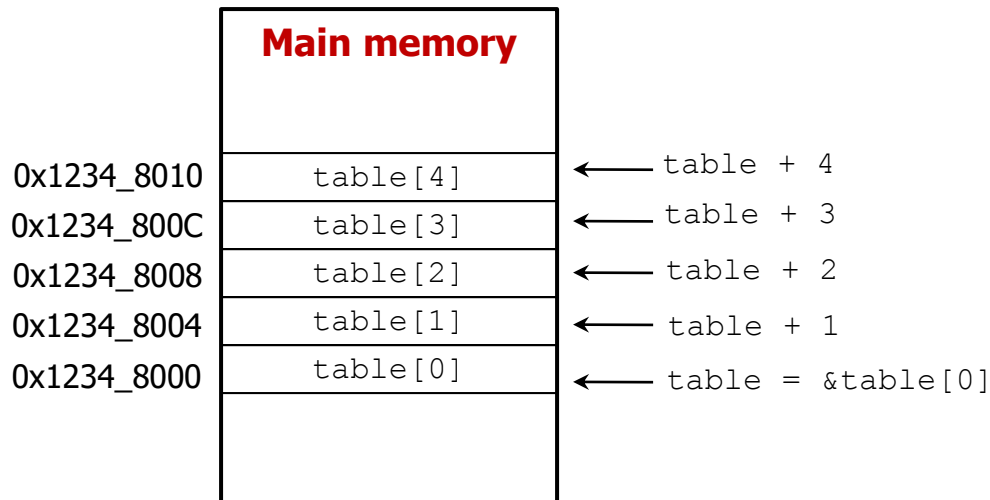
    addr_aa = &aa ;
    addr_bb = &bb ;

    printf("%c is stored as h'%x in memory\n", aa, *addr_aa);
    printf("%c is stored as h'%x in memory\n", bb, *addr_bb);
}
```

```
suhtw@suhtw-desktop:~/classes/comp212$ gcc char_dump.c -o char_dump
suhtw@suhtw-desktop:~/classes/comp212$ ./char_dump
A is stored as h'41 in memory
a is stored as h'61 in memory
suhtw@suhtw-desktop:~/classes/comp212$
```

Arrays

- Arrays are useful for accessing large amounts of similar data
 - Array element is accessed by index
 - Array size is the number of elements in the array
- Example
 - 5-element array of integers (`int table[5];`)
 - **Base address** = 0x12348000 (address of the first array element, `table[0]`)
 - First step in accessing an array: load base address into a register



```
int a ;  
a = *(table + 3)
```

compile

```
// $s0 = base address of the array  
// $t0 = a  
lw $t0, 12($s0);
```

Arrays vs Pointers

- Array indexing involves
 - Multiplying index by element size
 - Adding to array base address
- **Pointers** correspond directly to **memory addresses**

```
clear1(int array[], int size) {
    int i;

    for (i = 0; i < size; i += 1)
        array[i] = 0;
}
```

```

    move $t0,$zero    # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
    add $t2,$a0,$t1    # $t2 = &array[i]
    sw $zero, 0($t2)    # array[i] = 0
    addi $t0,$t0,1    # i = i + 1
    slt $t3,$t0,$a1    # $t3 = (i < size)
    bne $t3,$zero,loop1 # if (...)
                        # goto loop1
&array[0]
```

```
clear2(int * array, int size) {
    int * p;

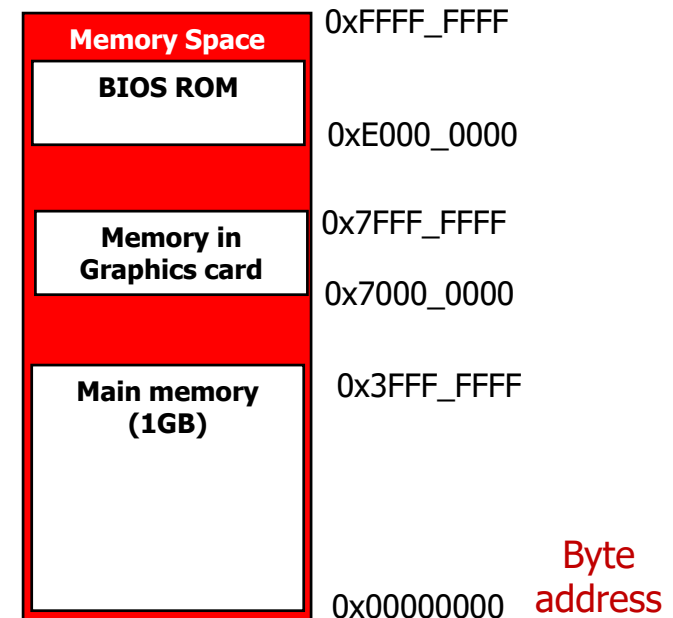
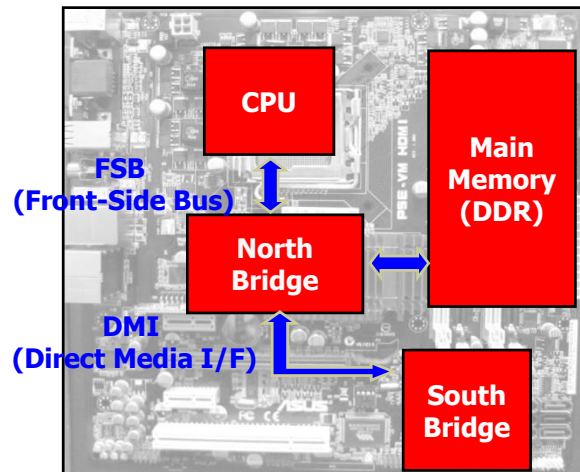
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

```

    move $t0,$a0    # p = & array[0]
    sll $t1,$a1,2    # $t1 = size * 4
    add $t2,$a0,$t1  # $t2 = &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
    addi $t0,$t0,4    # p = p + 4
    slt $t3,$t0,$t2    # $t3 = (p < &array[size])
    bne $t3,$zero,loop2 # if (...)
                        # goto loop2
&array[0]
```

Memory Map

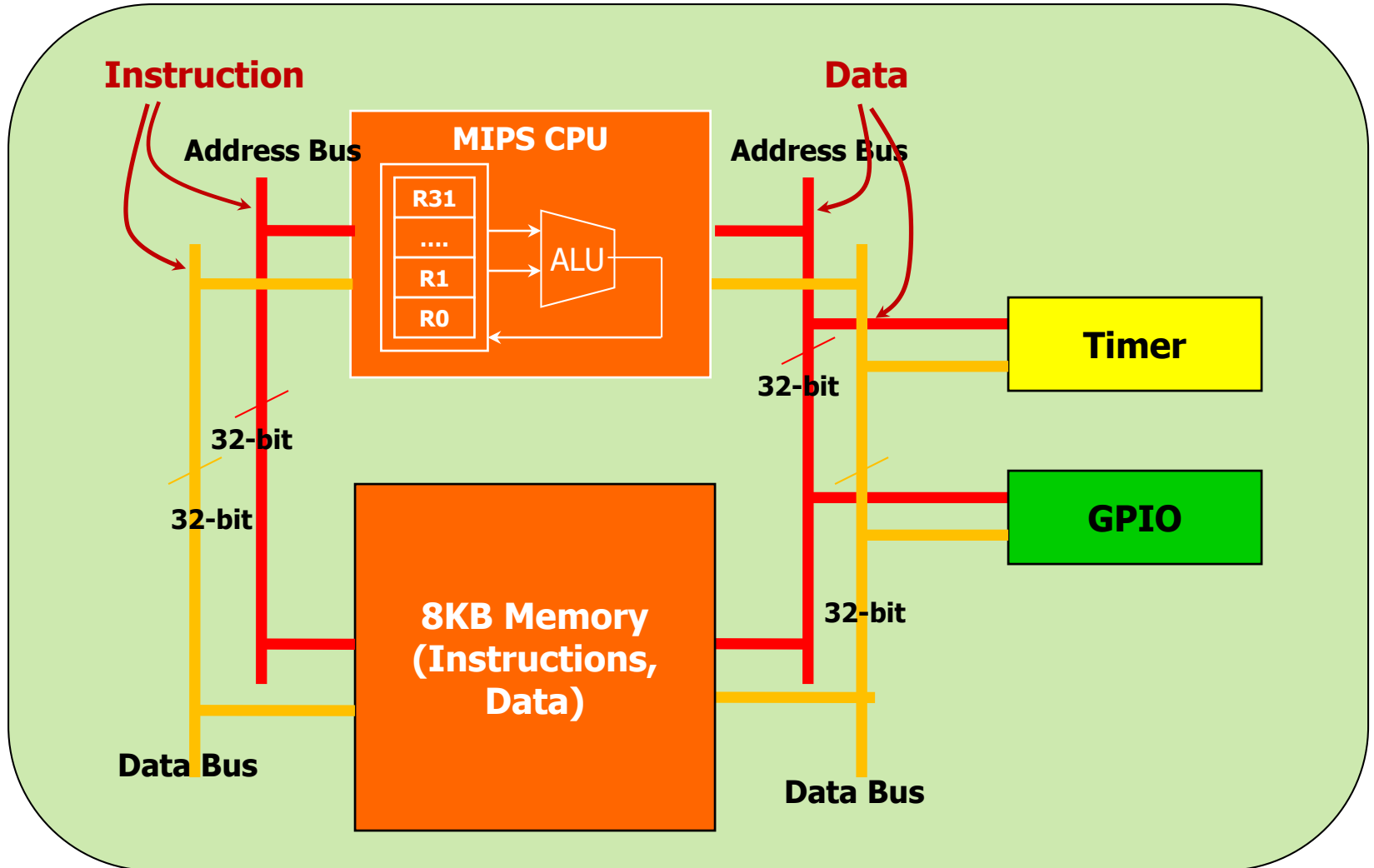
- **Memory map** indicates how memory space is laid out
 - Where the main memory and I/O devices are located in the memory space
 - **Memory-mapped I/Os**: when I/O devices are mapped in memory space, they are said to be memory-mapped I/Os
- Memory space is determined based on the size of the address bus
 - If the address bus is 32-bit wide, memory space is 4GB
 - If the address bus is 48-bit wide, memory space is 256TB



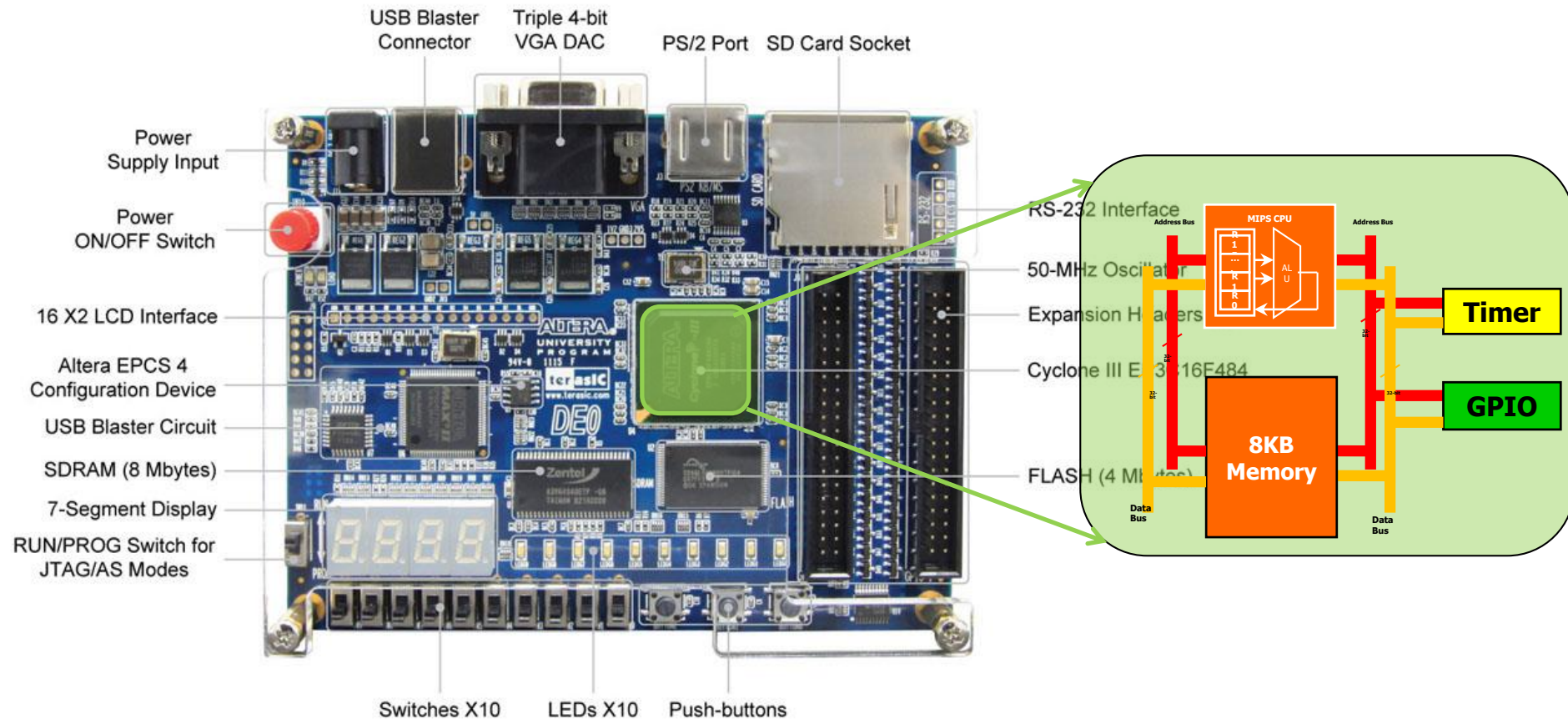
Our Computer System for Project

- We are going to use a simple computer system consisting of MIPS (not a complete MIPS), memory, and 2 peripheral devices (Timer and GPIO)
 - GPIO stands for General-Purpose I/O
 - Virtually all the peripheral devices have registers inside, so they can be programmed by software

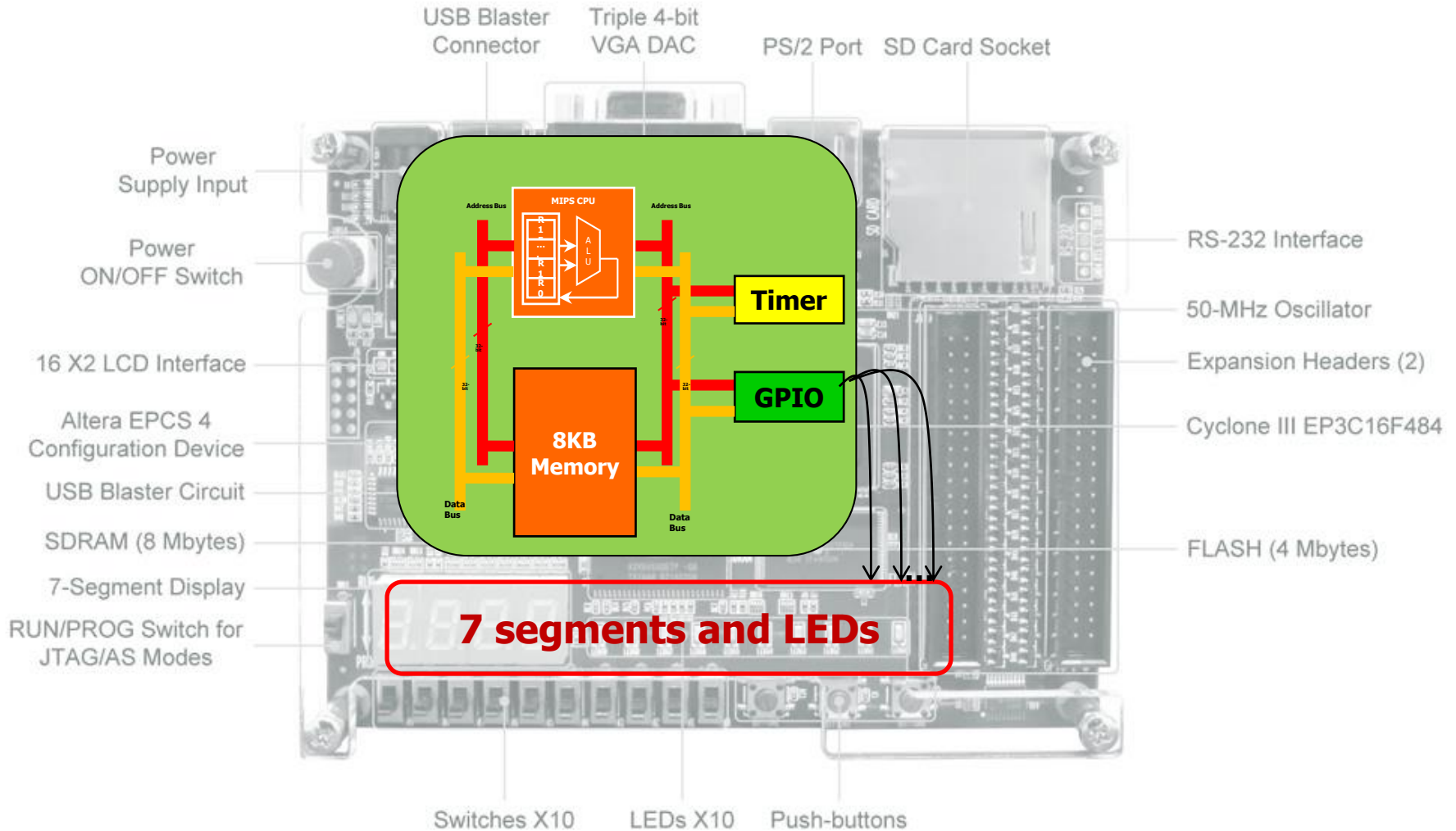
Our Computer System



Our Computer System in FPGA

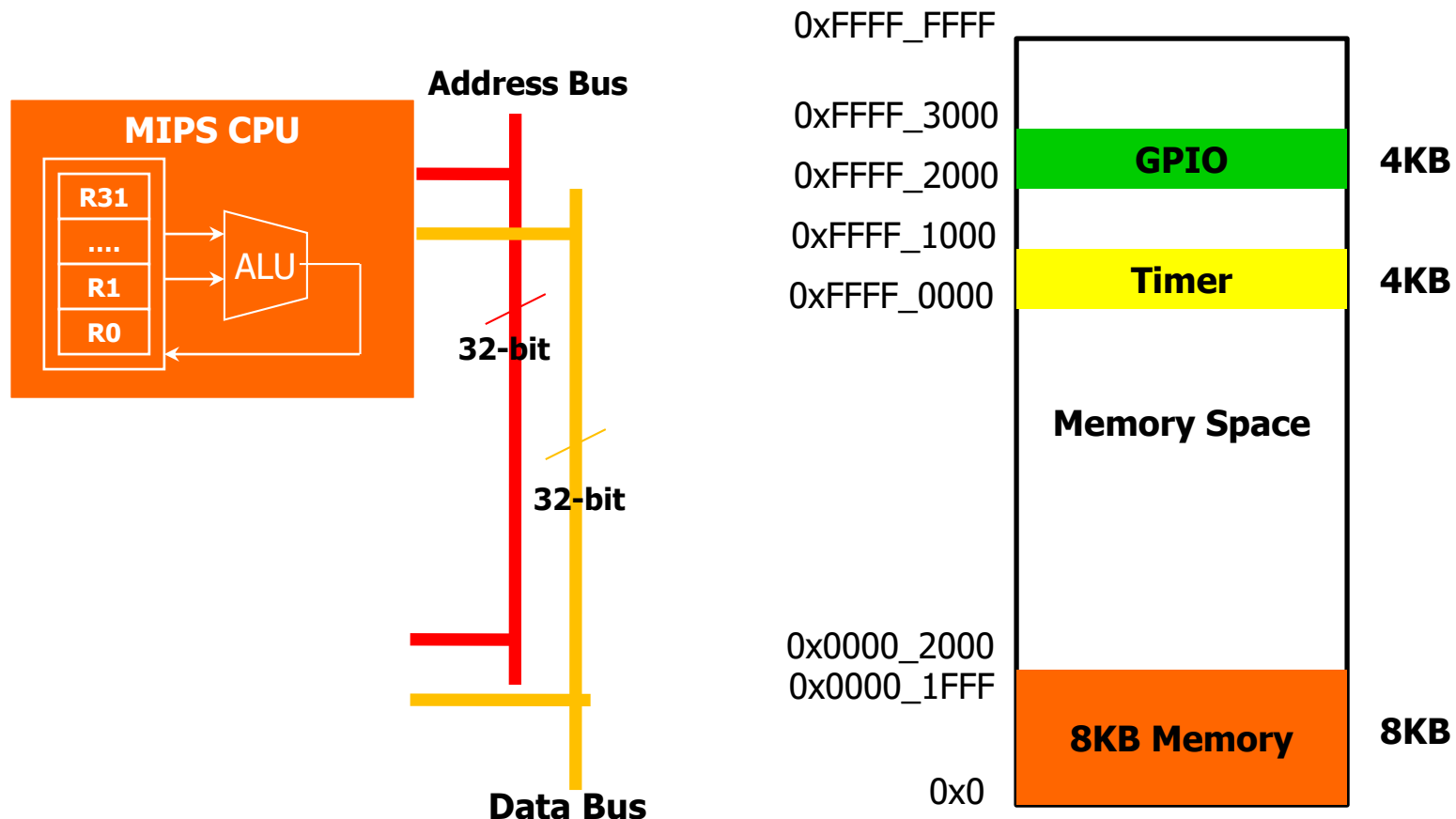


7 Segments and LEDs Connection



Memory Map of Our System

- Memory map of our system
 - You can change the map if you want by editing `Addr_decoder.v`



Linker Script

- Check out the linker script in `Makefile`
 - Figure out what the linker script says where the code and data in the program should be located in memory

`testvec.s`

```
.text
.align 4

la $sp, stack # load address
j SevenSeg

.data
.align 4
stack:
.space 1024
```

`testvec.lds`

```
SECTIONS
{
  . = 0x00000000;
  .text :
  {
    *(.text)
    *(.rodata)
  }

  . = ALIGN(1024);
  .data : {*(.data)}

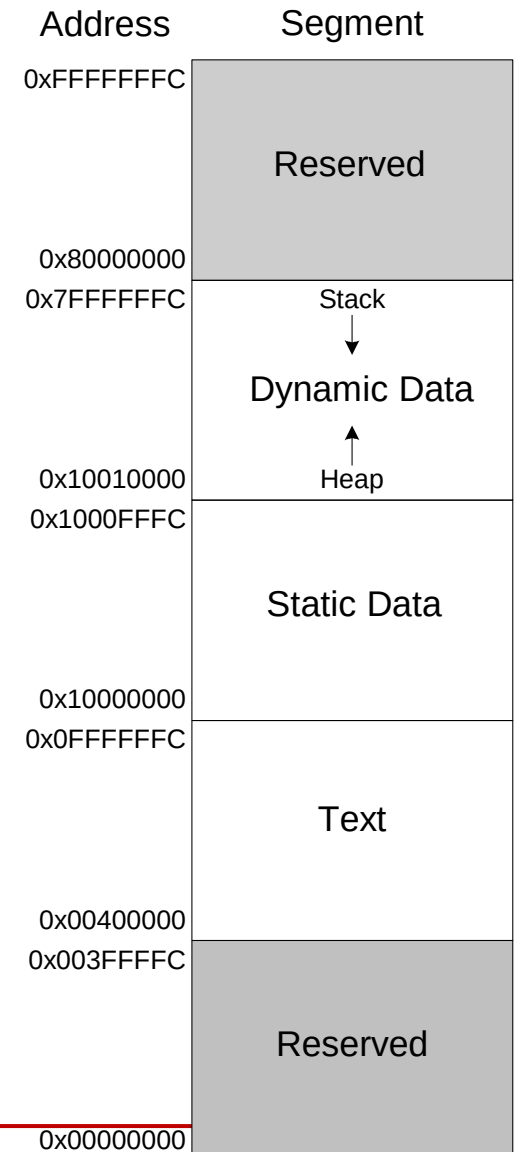
  .bss : {*(.bss)}

}
```

Backup Slides

The MIPS Memory Map

- Addresses shown are only a software convention (not part of the MIPS architecture)
- Text segment:** Instructions are located here
 - The size is almost 256MB
- Static and global data segment** for constants and other static variables
 - In contrast to local variables, global variables can be seen by all procedures in a program
 - Global variables are declared outside the main in C
 - The size of the global data segment is 64KB
- Dynamic data segment** holds **stack** and **heap**
 - Data in this segment are dynamically allocated and deallocated throughout the execution of the program
 - Stack** is used
 - To save and restore registers used by procedures
 - To hold local variables
 - Heap** stores data that is allocated by the program during runtime
 - Allocate space on the heap with `malloc()` and free it with `free()` in C
- Reserved segments** are used by the operating system



Example Program

```
int f, g, y; // global variables
```

```
int main(void)
{
    f = 2; //
    g = 3; //

    y = sum(f, g);
    return y;
}
```

```
int sum(int a, int b) {
    return (a + b);
}
```

compile

.data

f:

g:

y:

.text

main:

```
    addi $sp, $sp, -4    # stack frame
    sw   $ra, 0($sp)     # store $ra
    addi $a0, $0, 2      # $a0 = 2
    sw   $a0, f           # f = 2
    addi $a1, $0, 3      # $a1 = 3
    sw   $a1, g           # g = 3
    jal  sum              # call sum
    sw   $v0, y           # y = sum()
    lw   $ra, 0($sp)     # restore $ra
    addi $sp, $sp, 4      # restore $sp
    jr   $ra              # return to OS
```

sum:

```
    add  $v0, $a0, $a1    # $v0 = a + b
    jr   $ra              # return
```

Symbol Table of Example Program

- Assembler creates a **symbol table** that contains the names and their corresponding addresses of symbols (such as labels and global variable names)

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Executable of Example Program

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008
Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

```

addi $sp, $sp, -4
sw   $ra, 0($sp)
addi $a0, $0, 2
sw   $a0, 0x8000($gp)
addi $a1, $0, 3
sw   $a1, 0x8004($gp)
jal  0x0040002C
sw   $v0, 0x8008($gp)
lw   $ra, 0($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra

```

Example Program: In Memory

