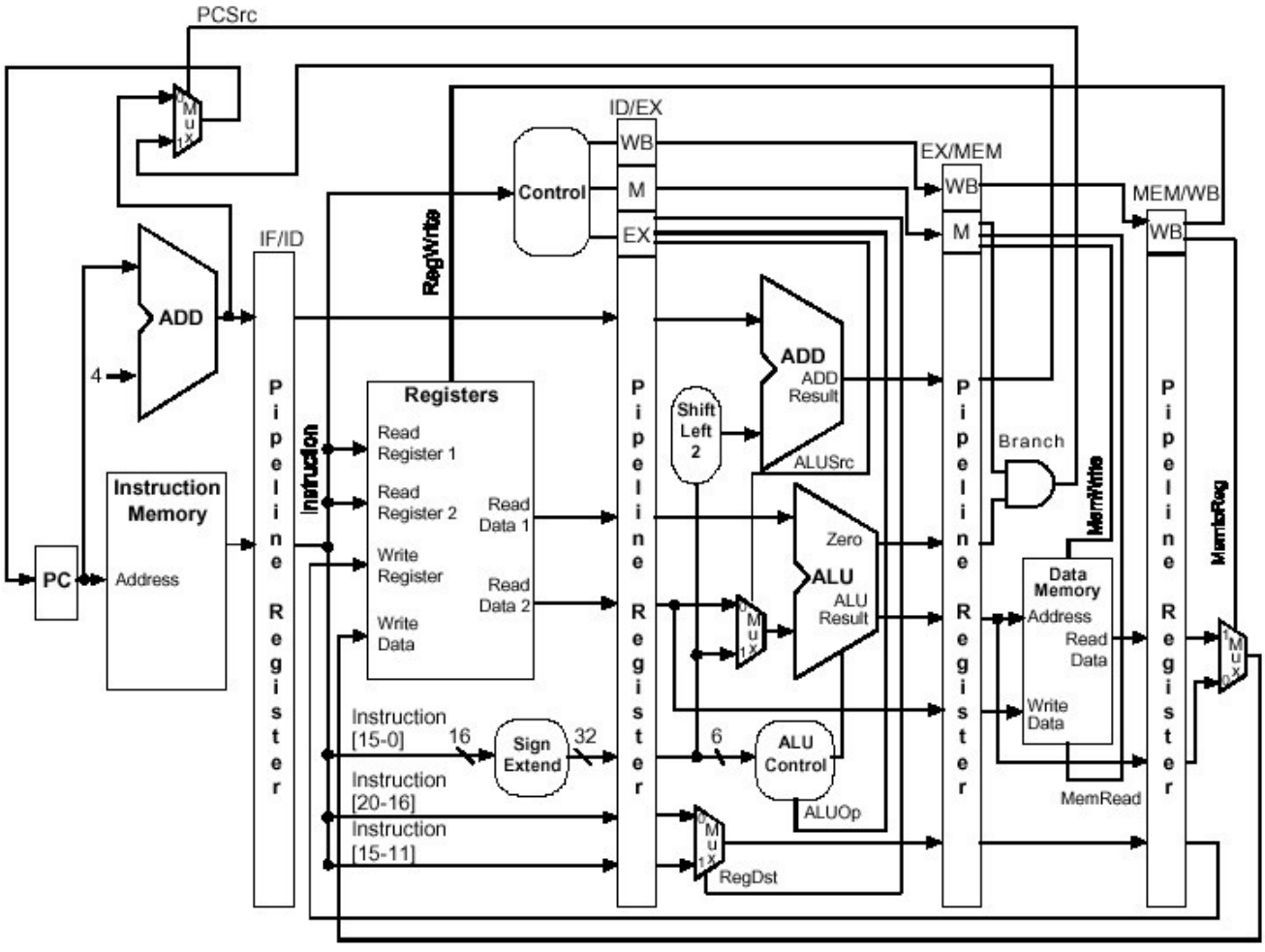
IPU Summer intensive course 2015 (Imamura, Kosuke 今村耕介)

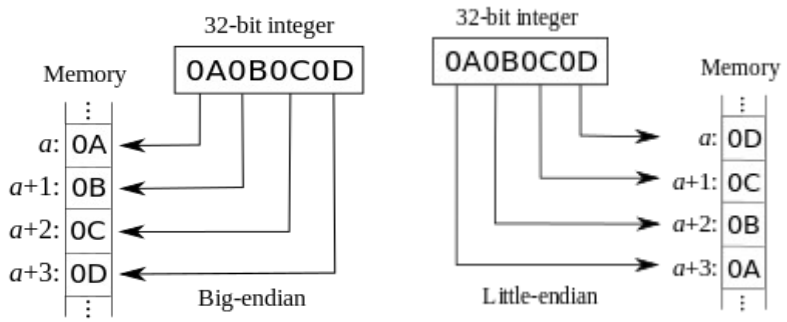
What makes difference between developing software on desktop type computers and embedded systems?

From booting the system and device interface all the way to high level support. We learn this through hand-on experience via building an interrupt driven I/O sub system, which you normally do not encounter in desktop application programming.

1. Learn MIPS

1.1 MIPS architecture





1.2 Instructions and programming

Assembly Language Syntax

Example:

```
        .data                   # data section
  item: .word 1                 # allocating one word, labeled (address)  'item'

        .text                   # text section (code section)
        .globl main             # main being global label
  main: lw $t0, item            # code
```

Strings are enclosed in double-quotes ("). Special characters in strings follow the C convention:

- newline \n
- tab \t
- quote \"

SPIM supports a subset of the assembler directives provided by the MIPS assembler:

.align n
: Align the next datum on a 2^n byte boundary.

.ascii str
: Store the string in memory, but do not null-terminate it.

.asciiz str
: Store the string in memory and null-terminate it.

.byte b1, ..., bn
: Store the n values in successive bytes of memory.

.data <addr>
: The following data items should be stored in the data segment. If the optional argument addr is present, the items are stored beginning at address addr.

.globl sym
: Declare that symbol sym is global and can be referenced from other files.

.half h1, ..., hn
: Store the n 16-bit quantities in successive memory halfwords.

.kdata <addr>
: The following data items should be stored in the kernel data segment. If the optional argument addr is present, the items are stored beginning at address addr.

.ktext <addr>
: The next items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, the items are stored beginning at address addr.

.label sym
Declare that symbol sym is a label.
.lcomm sym size
Allocate size bytes for symbol sym in the portion of the data segment that can be accessed via register $gp.
.space n
Allocate n bytes of space in the current segment (which must be the data segment in SPIM).
.set noat
Permit the program to refer to the $at register explicitly, and forbid SPIM from generating pseudoinstructions that modify $at.
.set at
Forbid the program from referring to the $at register explicitly, and permit SPIM to generate pseudoinstructions that modify $at (the default).
.text <addr>
The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, the items are stored beginning at address addr.
.word w1, ..., wn
Store the n 32-bit quantities in successive memory words.

Syscall

Function	Code in $v0	Argument or Return Value
PRINT_INT	1	$a0 = value
PRINT_FLOAT	2	$f12 = value
PRINT_DOUBLE	3	$f12 = value
PRINT_STRING	4	$a0 = address of string
READ_INT	5	Result placed in $v0
READ_FLOAT	6	Result placed in $f0
READ_DOUBLE	7	Result placed in $f0
READ_STRING	8	$a0 = address, $a1 = maximum length
SBRK (Mem allocate)	9	$a0 = number of bytes
EXIT	10	None
PRINT_CHAR	11	$a0 low byte = character
READ_CHAR	12	Character returned in low byte of $v0
OPEN_FILE	13	$a0 = address of filename, $a1 = flags, $a2 = mode. File descriptor returned in $v0. (negative if error occurred)
READ	14	$a0 = file descriptor, $a1 = address of buffer, $a2 = buffer length. Number of characters actually read returned in $v0.
WRITE	15	$a0 = file descriptor, $a1 = address of buffer, $a2 = number of bytes to write. Number of bytes actually written returned in $v0.
CLOSE	16	$a0 = file descriptor
EXIT2	17	$a0 = exit code

Table: MIPS registers and the convention governing their use.

Register Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and
v1	3	results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)
t3	11	Temporary (not preserved across call)
t4	12	Temporary (not preserved across call)
t5	13	Temporary (not preserved across call)
t6	14	Temporary (not preserved across call)
t7	15	Temporary (not preserved across call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved across call)
t9	25	Temporary (not preserved across call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp or s8	30	Frame pointer
ra	31	Return address (used by function call)

Data allocation examples

```
var1:     .word   3       # create a single integer variable with initial value 3
                          # C equivalent : unsigned int var1 = 3;

array1:   .byte   'a','b'         # create a 2-element char array with elements initialized
                                  #   to  a  and  b
                                  # C equivalent : unsigned char array[] = {'a', 'b'};


array2:   .space  40    # allocate 40 consecutive bytes, uninitialized
                        # a 40-element character array, or a 10-element integer array;
                        # C equivalent : unsigned char array[40];
                        # Or, unsigned int array[10];
```

NOTE: In assembly language programming, there is no signed or unsigned data type. It is programmer's responsibility to use proper instructions. Example: bgt, bgtu, ...

Example program 1 (Type the following program and run it on QtSpim)

```
        .data
  str:  .asciiz "Hello Iwate"

        .text
  main:
        addi $v0, $0, 4  # system call code for print_str
        la $a0, str      # address of string to print
        syscall          # print the string

        addi $v0, $0, 1  # system call code for print_int
        li $a0, 5        # integer to print
        syscall          # print it
        addi $v0, $0, 10    # v0 = $0 + 0
        syscall
```

Addressing Modes

MIPS is a load/store architecture, which means that only load and store instructions access memory. Computation instructions operate only on values in registers.

lw : Loads a word from a memory to a register. Address in memory must be word-aligned.
lb : Loads a byte from a memory to a register. Sign extends this result in the register.
lbu Loads a byte (unsigned) from a memory to a register. Zero extends in the register.
sw Store a word from a register to a memory. Address in memory must be word-aligned.
sb   Store the least significant byte of a register to a location in memory.

Example program 2

```
      .data
var1: .word   23                # declare storage for var1; initial value is 23

      .text
main:
     lw    $t0, var1            # load contents of RAM into register $t0:  $t0 = var1
```

```
    addi  $t1, $0, 5          # li $t1, 5   ("load immediate 5 to $t1")
    sw $t1, var1              # store contents of register $t1 into RAM:  var1 = $t1

    addi $v0, $0, 10          # v0 = $0 + 0
    syscall                   # exit service call (with v0=0)

NOTE:       lw    $t0, var1   # This will be assembled with two instructions as
                              # la    $at, var1       load address of var1 into $at
                              # lw    $t0, 0($at)     load word from address in $at + 0
```

Mode	Example	Effective address
Memory Direct	lw $t0, age	Address represented by label
Register Indirect	lw $t0, ($s3)	Contents of register in ()
Immediate Offset	lw $t0, 4($s3)	Contents of register in () + offset
Symbol Offset	lw $t0, list($s3)	Contents of register in () + address of label
Symbol + Immediate	lw $t0, list+4	Address of label + offset
Symbol + Immediate Offset	lw $t0, list+4($s3)	Address of label + offset + contents of register in ()

Immediate addressing can also be considered an addressing mode, but it is special in that it doesn't specify an address, but embeds the operand in the instruction code itself.
Examples:

```
Assume $t0 contains 1000
Address of "list" is 2000

lw  $a0, ($t0)          $a0 ← contents of address 1000
lw  $a0, 4($t0)         $a0 ← contents of address 1004
lw  $a0, list           $a0 ← contents of address 2000
la  $a0, list           $a0 ← 2000
lw  $a0, list+4         $a0 ← contents of address 2004
lw  $a0, list($t0)      $a0 ← contents of address 3000
lw  $a0, list+4($t0)    $a0 ← contents of address 3004
li  $a0, 5              $a0 ← 5
```

Example program 3

```
      .data
array1:       .space  12      #  12 bytes of storage to hold array of 3 integers

      .text
mail:
       la   $t0, array1    #  load base address of array into register $t0
       addi $t1, $0, 5     #  li    $t1, 5          #  $t1 = 5   ("load immediate")
       sw $t1, ($t0)       #  first array element set to 5; indirect addressing
       addi $t1, $0, 13    #  li $t1, 13           #   $t1 = 13
       sw $t1, 4($t0)      #  second array element set to 13
       addi $t1, $0, -7    #  li $t1, -7           #   $t1 = -7
       sw $t1, 8($t0)      #  third array element set to -7
```

Arithmetic Instructions

- most use 3 operands
- all operands are registers; no RAM or indirect addressing
- operand size is word (4 bytes)

examples

```
add   $t0,$t1,$t2      #  $t0 = $t1 + $t2;   add as signed (2's complement) integers
sub   $t2,$t3,$t4      #  $t2 = $t3 Ð $t4
addi  $t2,$t3, 5       #  $t2 = $t3 + 5;   "add immediate" (no sub immediate)
addu  $t1,$t6,$t7      #  $t1 = $t6 + $t7;   add as unsigned integers
subu  $t1,$t6,$t7      #  $t1 = $t6 + $t7;   subtract as unsigned integers

mult  $t3,$t4          #  multiply 32-bit quantities in $t3 & $t4, and store 64-bit
                       #  result in special registers Lo and Hi: (Hi,Lo) = $t3 * $t4
div   $t5,$t6          #  Lo = $t5 / $t6   (integer quotient)
                       #  Hi = $t5 mod $t6   (remainder)
mfhi    $t0            #  move quantity in special register Hi to $t0:   $t0 = Hi
mflo    $t1            #  move quantity in special register Lo to $t1:   $t1 = Lo
                       #  used to get at result of product or quotient

add   $t2,$0,$t3     #  move    $t2,$t3   $t2 = $t3
```

Control Structures

Branches

- comparison for conditional branches is built into instruction

```
b       target         #  unconditional branch to program label target
beq     $t0,$t1,target  #  branch to target if  $t0 = $t1
blt     $t0,$t1,target  #  branch to target if  $t0 < $t1
ble     $t0,$t1,target  #  branch to target if  $t0 <= $t1
bgt     $t0,$t1,target  #  branch to target if  $t0 > $t1
bge     $t0,$t1,target  #  branch to target if  $t0 >= $t1
bne     $t0,$t1,target  #  branch to target if  $t0 <> $t1
```

Jumps

```
j       target  #  unconditional jump to program label target
jr      $t3           #  jump to address contained in $t3 ("jump register")
```

Subroutine Calls

subroutine call: "jump and link" instruction

```
jal     sub_label       #  "jump and link"
```

- copy program counter (return address) to register $ra (return address register)
- jump to program statement at sub_label

subroutine return: "jump register" instruction

```
jr      $ra     #  "jump register"
```

- jump to return address in $ra (stored by jal instruction)

Note: return address stored in register $ra; if subroutine will call other subroutines, or is recursive, return address should be copied from $ra onto stack to preserve it, since jal always places return address in this register and hence will overwrite previous value

```
slt  $rd, $rs, $rt      # R[d] = R[s] < R[t] ? 0^30::1 : 0^32
sltu $rd, $rs, $rt      # R[d] = R[s] < R[t] ? 0^30::1 : 0^32
```

slt $rt, $rs, immed   # R[d] = R[s] < immed ? $0^{30}::1 : 0^{32}$
sltu $rt, $rs, immed   # R[d] = R[s] < immed ? $0^{30}::1 : 0^{32}$

Notice I'm using the conditional expression (question-mark/colon operator) in C to explain the semantics.

Here are the differences between all the instructions.

- **slt** performs comparisons of two registers, assuming they store 32-bit 2C representations.
- **sltu** performs comparisons of two registers, assuming they store 32-bit UB representations.
- **slti** performs comparisons of a register and a sign-extended immediate value, assuming the register stores 32-bit 2C representations.
- **sltiu** performs comparisons of a register and a zero-extended immediate value, assuming the register stores 32-bit UB representations.

As you might guess, **slt** and **sltu** are R-type instructions, while **slti** and **sltiu** are I-type instructions.

Translated Pseudoinstructions

Here's the table for translating pseudoinstructions.

Pseudoinstruction	Translation
bge $rt, $rs, LABEL	slt $t0, $rt, $rs beq $t0, $zero, LABEL
bgt $rt, $rs, LABEL	slt $t0, $rs, $rt bne $t0, $zero, LABEL
ble $rt, $rs, LABEL	slt $t0, $rs, $rt beq $t0, $zero, LABEL
blt $rt, $rs, LABEL	slt $t0, $rt, $rs bne $t0, $zero, LABEL

where we assume **$t0** is not **$rs**, nor **$rt** (if it is, pick another register that's being unused), and **$zero** is **$r0**.

Example program 4

```
# a MIPS code fragment to increment each element of
# an array of 100 integers.

.data
array: .space 400

.text
main:
      addi   $8, $0, 0      # value 0 goes into $8 (a counter)
      addi   $9, $0, 100    # $9 is ending value of counter
      la     $11, array     # $11 is pointer into array of integers
loop_top:
      beq    $8, $9, done_loop
      lw     $10, 0($11)    # get array element
      addi   $10, $10, 1    # add one to it
```

```
sw     $10, 0($11)    # put it back
addi   $11, $11, 4    # update pointer
addi   $8, $8, 1      # update counter
beq    $0, $0, loop_top  # uncondional branch
```

Some notes:
la is not really a MIPS R2000 instruction. It is an example of and instruction that is allowed in assembly language code, but is translated into MIPS R2000 code by the assembler. This makes the job of writing this assembly language code easier.

For this la instruction, the assembler produces

(if symbol array is assigned address 0x00aa0bb0)

```
lui  $11, 0x00aa            # $11 gets value 0x 00aa 0000
ori  $11, $11, 0x0bb0       # $11 gets value 0x 00aa 0bb0
```

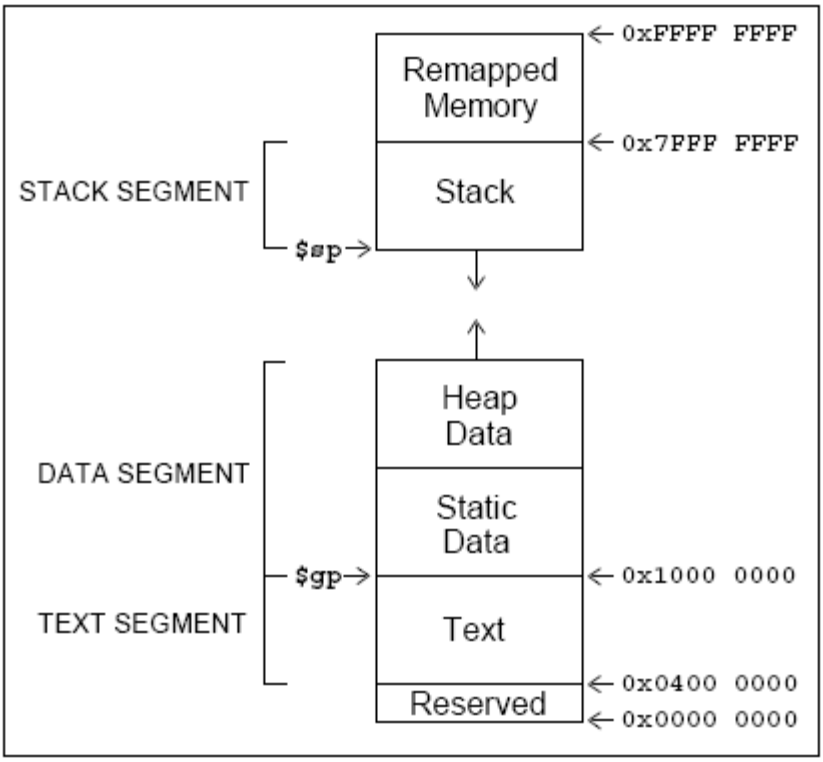
Subroutine

```
        .text
main:
      .....
        jal     subprog
        jr      $ra


subprog:          # Beginning of subprogram

        jr      $ra    # End of subprogram
```

Parameter Passing
Register conventions

r0	($zero)	- constant 0 (hardware)
r1	($at)	- reserved for assembler
r2-r3	($v0-$v1)	- expression evaluation and function return value
r4-r7	($a0-$a3)	- four arguments
r8-r15	($t0-$t7)	- eight temporaries (not preserved across calls)
r16-r23	($s0-$s7)	- eight saved temporaries (preserved across calls)
r24-r25	($t8-$t9)	- two temporaries (not preserved across calls)
r26-r27	($k0-$k1)	- reserved for OS kernel
r28	($gp)	- pointer to global area
r29	($sp)	- stack pointer
r30	($fp)	- frame pointer
r31	($ra)	- return address (hardware)

Parameter passing via stack



--- caller ---
addi $sp, $sp, 4
sw
jal xyz
addi $sp, $sp, 4

---- subroutine ---

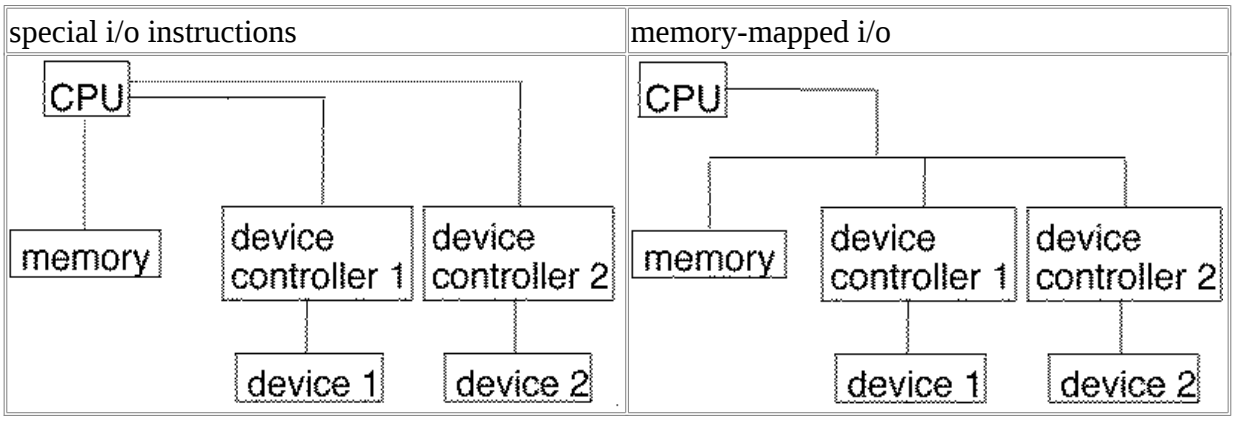
```
xyz:                        # at the time of call   parameter is in 0($s0)
     addi $sp, $sp, -4   # space to save register $s0   (sp moved down by 4)
     sw   $s0, 0($sp)    # save $s0; s0 can be used freely
     lw   $s0, 4($sp)     # retrieve the passed parameter on the stack; 4 up from $sp
      ....
      ....
     lw   $s0, 0($sp)    # restore $s0
     addi $sp, $sp, 4    # adjust  $sp
     addi $v0, $0, 1     # return value is set to 1
     jr    $ra
```

Note: How do you call a subroutine from a subroutine?

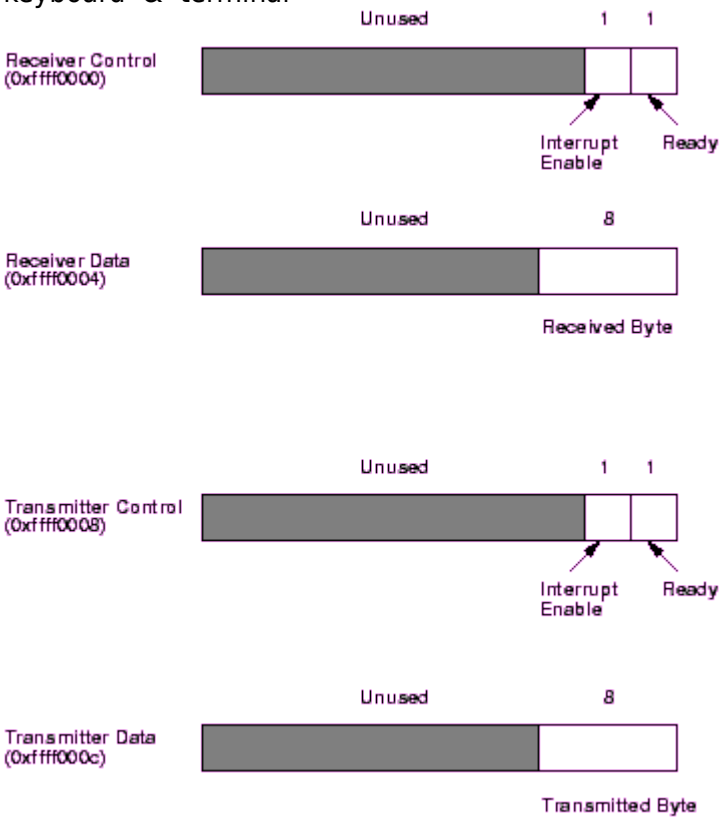
Hint: The “jal” writes to $ra to store return address, so you have to save $ra before you do “jal”, then restore $ra.

2. Device interface

Older computers had special instructions for i/o. In contrast, the MIPS, like most modern machines uses *memory-mapped i/o*. A portion of the address space is dedicated to communication paths to input or output devices; input is done via a load from an address in this space, and output is done with a store. The two organizations are displayed below.



keyboard & terminal



One way to do i/o is by *polling*. The processor reads from the control register in a loop, waiting for the corresponding device to set its "ready" bit. The processor then loads from (input) or stores into (output) the associated data register. This load or store unsets the ready bit temporarily until the i/o operation is completed. Code fragments to read and

write a character using polling appear below.

Polling

character input into $v0	*output of character in $a0*

`lui $t0,0xffff #ffff0000`
`waitloop:`
`lw $t1,0($t0) #control`
`andi $t1,$t1,0x0001`
`beq $t1,$0,waitloop`
`lw $v0,4($t0) #data`
	`lui $t0,0xffff #ffff0000`
	`waitloop:`
	`lw $t1,8($t0) #control`
	`andi $t1,$t1,0x0001`
	`beq $t1,$0,waitloop`
	`sw $a0,12($t0) #data`

3. Interrupt

Interrupts Initiated outside the instruction stream Arrive asynchronously (at no specific time), Example:

I/O device status change
I/O device error condition

Traps Occur due to something in instruction stream.

Examples:

Unaligned address error
Arithmetic overflow
System call

These registers are part of coprocessor 0's register set and are accessed by the lwc0, mfc0, mtc0, and swc0 instructions.

Coprocessor 0 register numbers (SPIM)

Register Name	Number	Usage
BadVAddr	8	Memory address at which address exception occurred
Count	9	Timer count, Increments every 10 ms
Compare	11	Interrupt when Count == Compare
Status	12	Interrupt mask and enable bits
Cause	13	Exception type and pending interrupt bits
EPC	14	Address of instruction that caused exception

Exception Code

Number	Name	Description
0	INT	External interrupt
4	ADDRL	Address error exception (load or instruction fetch)

5	ADDRS	Address error exception (store)
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data load or store
8	SYSCALL	Syscall exception
9	BKPT	Breakpoint exception
10	RI	Reserved instruction exception
12	OVF	Arithmetic overflow exception
13	TR	Trap
15	FLE	Floating point exception

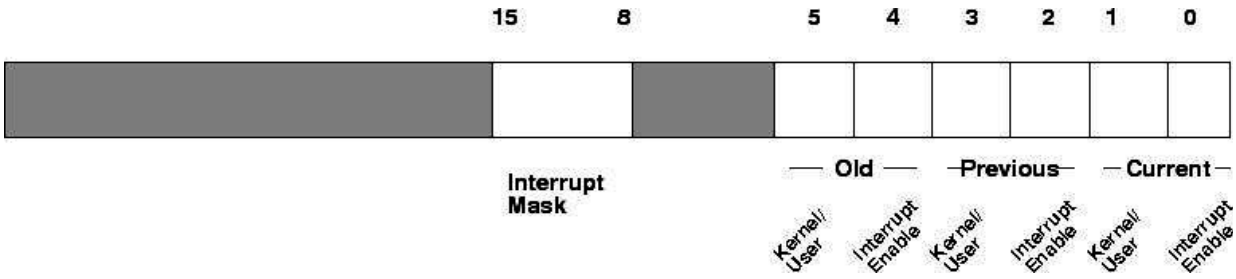


Figure: The `Status` register.

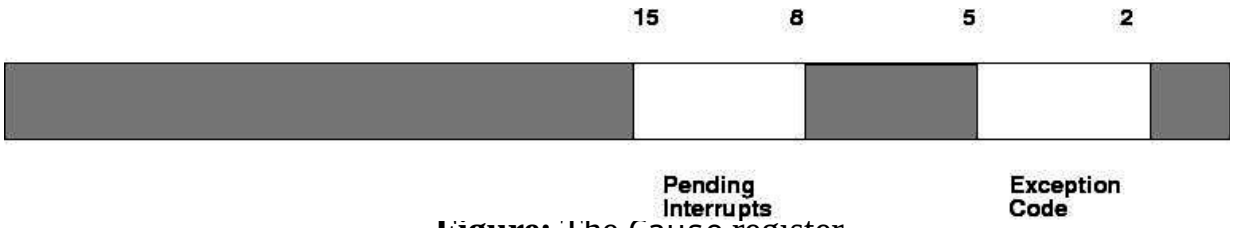


Figure: The Cause register.

Before you run the program:

1. QtSpim: Menu -> Simulator -> setting : Uncheck Load Exception Handler. Because, we are writing our own.

2. QtSpim: Menu -> Simulator -> setting : Check Enable Mapped IO

QtSPIM comes up with the status to be 3000FF10 as default.
That is : 0011 0000 0000 0000 1111 1111 0001 0000

That means, QtSpim's interrupu mask is all 1, thus everyone can interrupt. The user mode bit 4 is 1. Interrput disabled.

What needs to be done to get a system up running.
1. Enable the devices' interrupt enable bit.
2. Enable CPU's interrupt bit to accept interrupts.

3. Now, everything is ready, so run a user program by jumping to main.

When an interrupt occurs, EPC ← PC ; Cause Register ($13) ← cause ; PC ← 0x80000180

Interrupt Service Routine

```
# kernel data section
.kdata
temp: .space 80

# MIPS exception vector is 0x80000180
.ktext 0x80000180
# save $at first
add $k0, $0, $at

# (step 1) save registers that you are going to use but $k0, $k1
la $k1, temp
sw $k0, 0($k1)
sw $a0, 4($k1)
sw $a1, 8($k1)

...

# c0$13       = 32768 ( 1000 0000 0000 0000 ) timer
#             =   2048 ( 0000 1000 0000 0000 ) key hit
#             =   1024 ( 0000 0100 0000 0000 ) terminal output

mfc0 $k0, $13          # Move Cause into $k0
srl    $a0, $k0, 2       # Extract Exception Code field
andi $a0, $a0, 0xf    # $a0 contains  Exception Code
bgtz $a0, done         # Branch if ExcCode is Int (0)
add $a0, $0, $k0      # Move Cause into $a0
mfc0 $a1, $14          # Move EPC into $a1 , in case it's trap and need to do EPC= EPC+4

# check $a0
# if trap (software exception then process it and EPC = $a1 + 4
# else it is hardware interrupt request, process it

la $k1, temp           # restore registers
lw $k0, 0($k1)
lw $a0, 4($k1)
lw $a1, 8($k1)

mtc0 $0 $13        # Clear Cause register
mfc0 $k0 $12       # Set Status register
ori  $k0 0x1       # Interrupts enabled
mtc0 $k0 $12
eret                   # return from exception
```

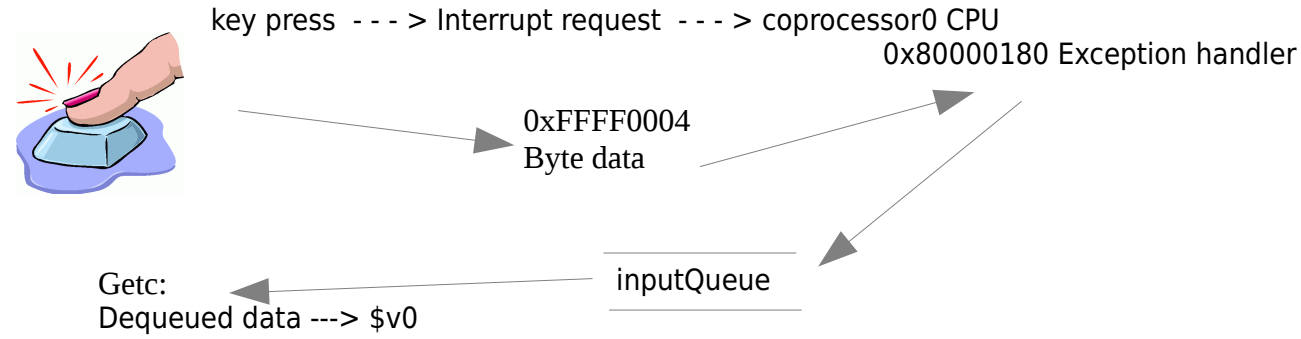
--- simple timer example, but it is complete ---

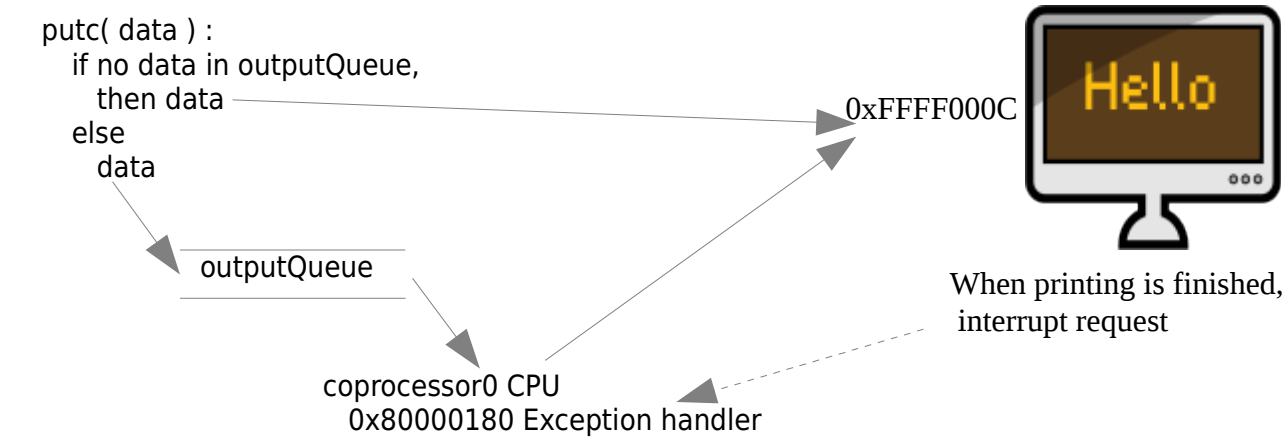
```
# minimalistically simple example of timer interrupt

        .text
        .globl main
main:
    mfc0 $t0, $12
    ori $t0, $t0, 0xff01   # enable interrupt
    mtc0 $t0, $12

    li $t0, 10       # compare register value
    mtc0 $t0, $11   #
    mtc0 $zero, $9  # init counter
                # interrupt occurs when $9 == $11

    xor $a0, $a0, $a0

again:
    addi $a0, $a0, 1
    j again

    jr $ra

#--- END MAIN


        .kdata
temp:   .space 16


        .ktext 0x80000180
timer_handler:
        la $k0, temp
        sw $a0, 0($k0)
        sw $v0, 4($k0)
        sw $t0, 8($k0)
        sw $t1,12($k0)

        addi $a0, $0, 0x30
        lui     $t0,0xffff      #ffff0000
waitloop:
        lw      $t1,8($t0)      #control
        andi    $t1,$t1,0x0001
        beq     $t1,$0,waitloop
        sw      $a0,12($t0)     #data


        mtc0 $zero, $9

        lw $a0, 0($k0)
        lw $v0, 4($k0)
        lw $t0, 8($k0)
        lw $t1,12($k0)
```

```
mtc0 $0, $13        # Clear Cause register
mfc0 $k0, $12          # Set Status register
ori  $k0, 0x1        # Interrupts enabled
mtc0 $k0, $12
eret
```

4. Interrupt driven I/O

getc()





Pseudo Code Exception Handler (at 0x80000180)

1. Check the cause register
2. If cause is keyboard interrupt
   inq_count++;
   inputQueue [inq_tail_index] = data;
   inq_tail_index = (inq_tail_index + 1) & (2^n -1);

3. If cause is terminal interrupt
   outq_count--;
   data = outputQueue [outq_head_index];
   outq_head_index = (outq_head_index + 1) & (2^n -1);

System Initialization code

```
__start:
   Enable keyboard interrupt (Device)
    Enable terminal interrupt (Device)
    Set interrupt mask (CPU)
    Enable interrupt (CPU)
    jump to main

main:
   ch = getc( );
    putc( 0x0A );
    putc( ch );
    jump to main

getc:
   if inq_count>0
          data = inputQueue [ inq_head_index ];
          Mask keyboard interrupt
          inq_count--;
          Unmask keyboard interrupt
          inq_head_index = ( inq_head_index + 1 ) & N      (N = 2^n -1)
          dequeue it. Dequeued data ---> $v0
    else
           suspend_process()  # this is if multitasking system, blocking

-------------------------------------------------------------------------------------------------

putc( ):
   get data from the stack
   if outq_count <   2^n
          outputQueue [ outq_tail_index ];
          Mask keyboard interrupt
          outq_count++;
          Unmask keyboard interrupt
          outq_tail_index = ( outq_tail_index + 1 ) & N      (N = 2^n -1)
    else
         ....
```

5. Shared Memory Programming

---- creat shared memory ----

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

main() {
key_t key;
```

```
int shm_id;
char *shm;
char *message = "hello";

//key=0x520260A;
key = ftok("/home/kosuke/IPU", 'a' );
printf("key= %x\n", key);
shm_id = shmget(key, sizeof(char)*10,  IPC_CREAT| 0666);

printf(" shm_id = %d\n", shm_id);
getchar();

shm = shmat(shm_id,0,0);
printf(" shm = %d\n", shm);
getchar();
sprintf(shm,"%s",message);
}
```

----- READING SHARED MEMORY -----

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

main() {

int shm_id;
void *shm;
char *message;

//key_t key=0x520260A;
key = ftok("/home/kosuke/IPU", 'a' );
printf("key= %x\n", key);

message = malloc(sizeof(char) * 10);
shm_id = shmget(key, sizeof(char) * 10, NULL);
shm = shmat(shm_id,NULL,NULL);
if(shm == NULL)
{
printf("error");
}
sscanf(shm,"%s",message);
printf("\n message = %s\n",message);
}
```