Interruption

MU4IN106 Multi

Plan

An interrupt is a mechanism that allows the hardware to force the software to do urgent processing.

The goal of this course is to study the communication between hardware and software using the interrupt signals.

- Hardware view
  → Who generates the interrupt signals and how are they routed to the MIPS processors?
- Software view
  → What does the software do when a MIPS processor receives an interrupt signal

Hardware point of view

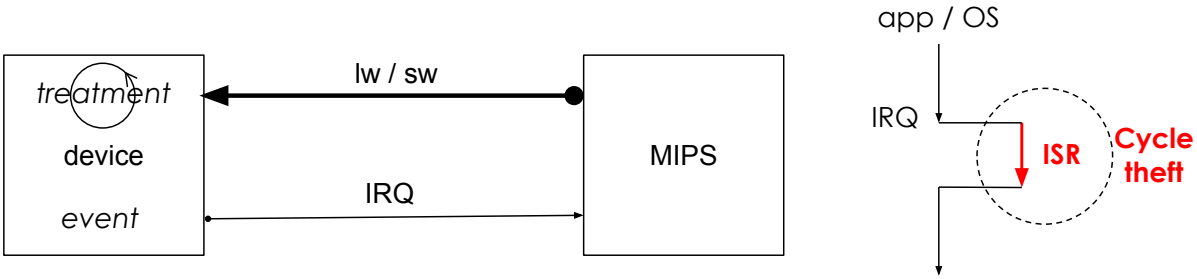
Memory mapped devices

- A device is a component that can be controlled by registers
  - Device registers are not MIPS registers!
- Registers are placed in the MIPS address space
  - each register has an address aligned to a word
  - the addresses are in the regions accessible in kernel mode
- The address segments are aligned on powers of 2 words
  - This simplifies the decoding because it is sufficient to use the most significant bits in the address and not an arithmetic comparison
- Devices can be multi-channel
  - we have here a multi-TTY, a multi-timer, a muti-ICU
  - each device has as many register cards as channels

Type of devices

- Devices receive commands from the OS
  - (with sw) to configure or transmit data
  - (with lw) to read the status or retrieve data.
- The devices are always **targets** since they respond to these commands..
- But, some devices can also issue read and write commands on their own behalf and are then also **initiators**.
  This is not the case today

IRQ Interrupt ReQuest

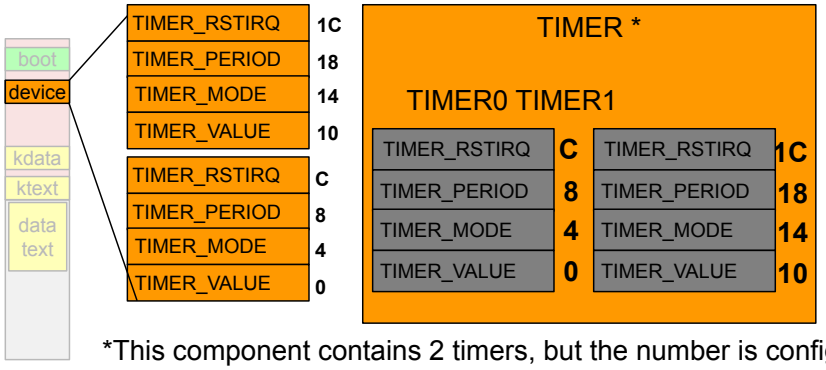


- IRQ is a boolean signal allowing the device to inform the OS of the end of a command
- It causes suspension of the current program ⟶ theft of cycles
- And the execution of an uninterruptible and time-bound Interrupt Service Routine (ISR)
  - read the records most often
  - acknowledge the IRQ ⟶ ask the device to lower the IRQ

TIMER

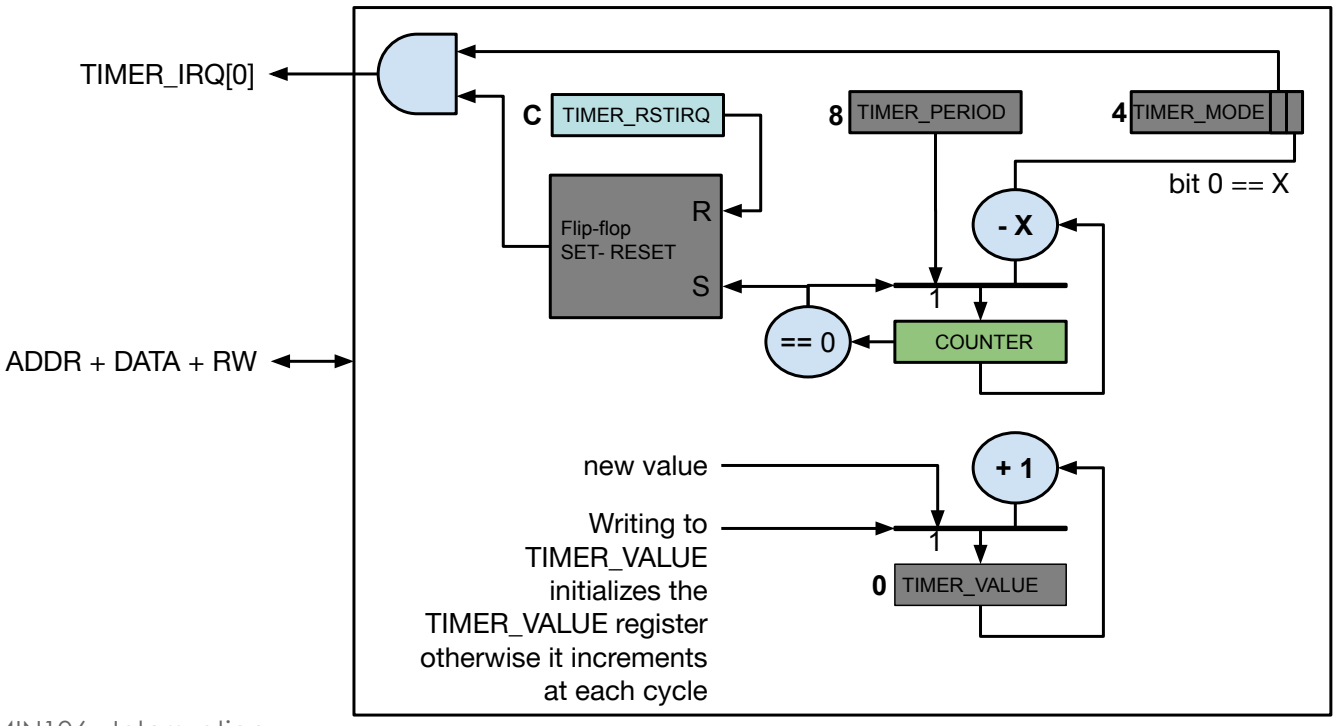
The TIMER contains time counters that can raise periodic interrupts.
It is a target device controlled by read/write accesses in its registers.

- TIMER_VALUE (read/write) +1 at each cycle
- TIMER_MODE (write only) configures the operating mode
  bit 0 : 1 = timer on (countdown); 0 = timer off
  bit 1 : 1 IRQ requested ; 0 IRQ masked
- TIMER_PERIOD (write only) period between 2 IRQ
- TIMER_RESETIRQ (write only) writing to this address acknowledges the IRQ

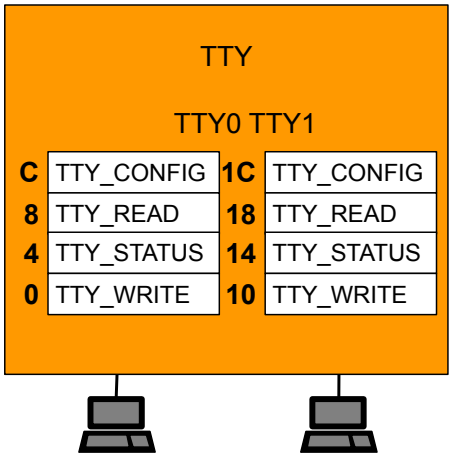


*This component contains 2 timers, but the number is configurable.

TIMER



TTY terminal controller



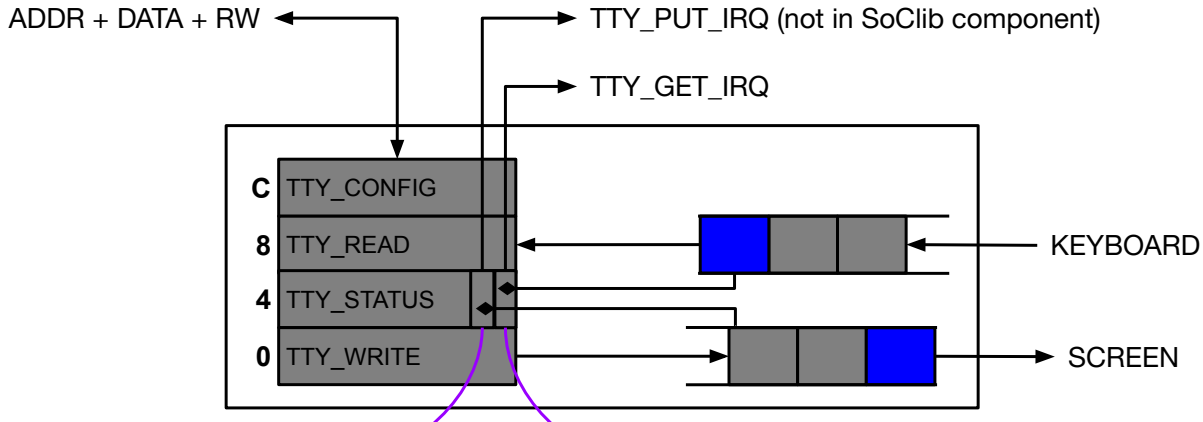
All registers are word aligned,
each terminal uses a segment of 4 words.

For each controlled terminal

- TTY_WRITE 1 byte write only
  output to the screen
- TTY_STATUS 1 byte read only
  ≠ 0 if there is a pending character
  in TTY_READ
- TTY_READ 1 byte read only
  character typed on keyboard
- TTY_CONFIG not used in this version
  allows the configuration of
  e.g. data throughput

Each TTY raises a single IRQ if a character is received
But usually, there is another one when a character can be sent!

TTY terminal controller



to 1 if there is a free space
in the output FIFO

TTY_PUT_IRQ is active
if TTY_STATUS [1] == 1

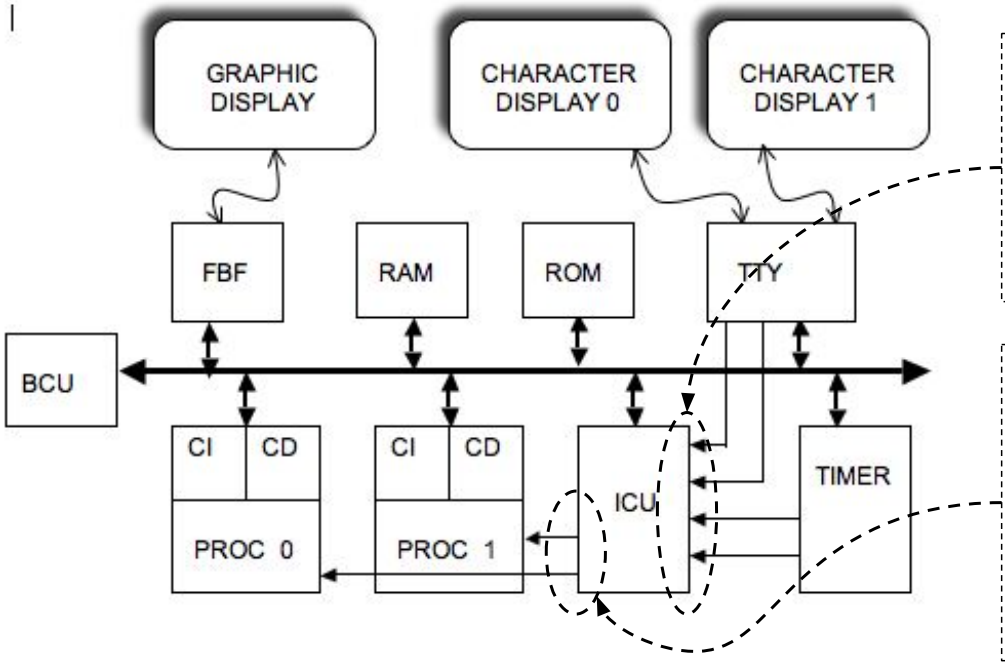
To acknowledge TTY_PUT_IRQ
you have to write in TTY_WRITE

to 1 if there is a character present
in the input FIFO

TTY_GET_IRQ is active
if TTY_STATUS [0] == 1
To acknowledge TTY_GET_IRQ

you have to read in TTY_READ

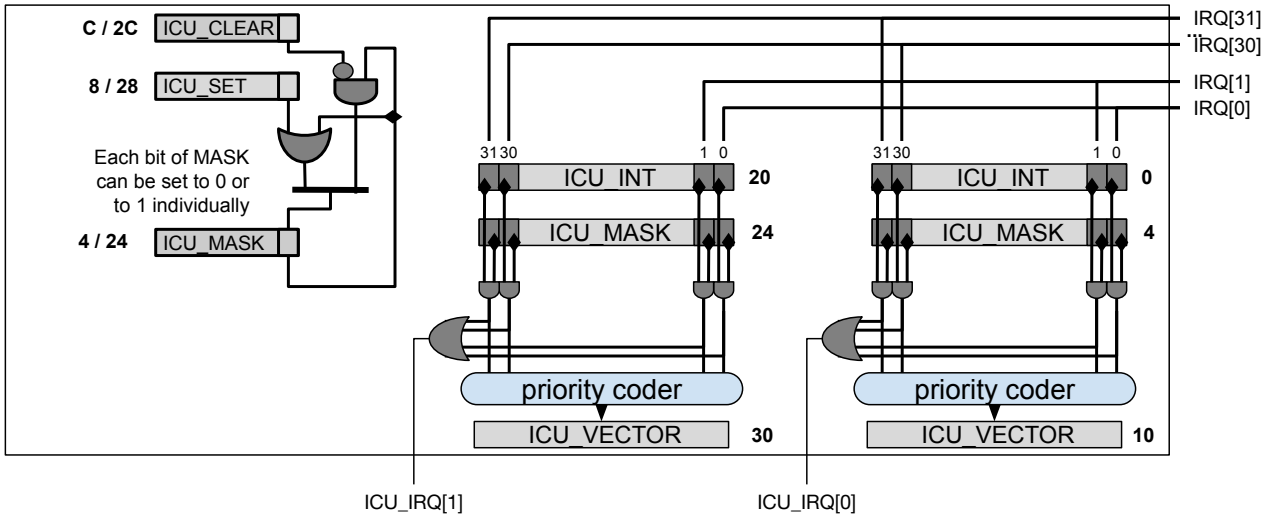
IRQ routing in the SoC



The IRQ connection is defined in the `top.cpp`
It is not modifiable by the OS

IRQ distribution to the proc is configured by the ICU masks
It is modifiable by the OS

Interrupt Controler Unit ICU

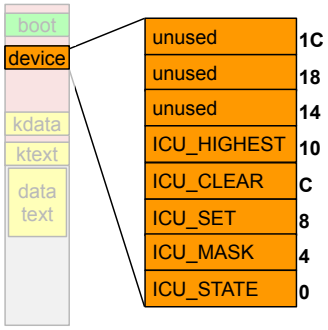


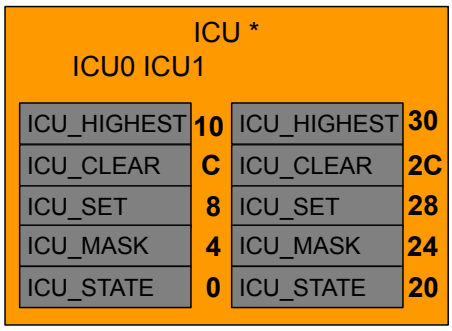
It is absolutely necessary to configure the ICUs so that an IRQ is "routed" to a single MIPS

Interrupt Controler Unit ICU

The ICU is a multi-concentrator of IRQ signals. Each IRQ can be masked.
In the case of a multicore architecture, it allows to route IRQs to any MIPS.
It is a target device controlled by read/write accesses in its registers.

- ICU_STATE (read only) status of IRQ lines
- ICU_MASK (read only) IRQ line masks
- ICU_CLEAR (write only) command to set IRQ masks to 0
- ICU_SET (write only) command to set IRQ masks to 1
- ICU_HIGHEST (read only) number of the highest priority active IRQ line





* The drawn component contains 2 ICUs, the number is configurable.

If there is no active IRQ the ICU_HIGHEST register contains 32,

This register is named ICU_IT_VECTOR in this code

Accessible address space

- The address space is the set of addresses that can be produced by the MIPS processor, only a part of it is accessible.
- This is a hardware choice defined in `top.cpp` and in `seg.ld`

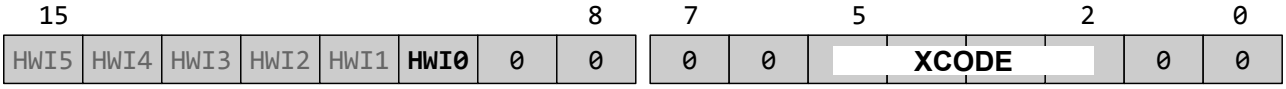
```
seg_reset_base  = 0xBFC00000;

seg_kcode_base  = 0x80000000;
seg_kunc_base   = 0x81000000;
seg_kdata_base  = 0x82000000;

seg_code_base   = 0x00400000;
seg_data_base   = 0x01000000;
seg_stack_base  = 0x02000000;

seg_tty_base    = 0x90000000;
seg_timer_base  = 0x91000000;
seg_ioc_base    = 0x92000000;
seg_dma_base    = 0x93000000;
seg_gcd_base    = 0x95000000;
seg_fb_base     = 0x96000000;
seg_icu_base    = 0x9F000000;
```

- Addresses ≥ 0x80000000 are accessible only in kernel mode
- Note a non-cached segment to compensate for the lack of coherence of L1 caches
- Note the absence of stack segments for the kernel
- Note that the device registers are in the area managed by the kernel

Software view

System registers: Status, Cause, EPC

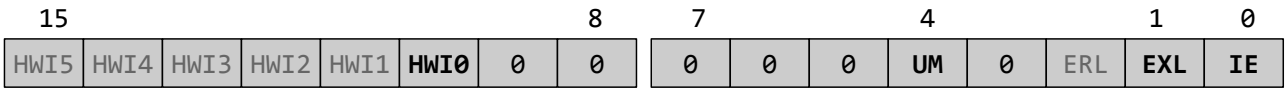
The register `c0_cause ($13)` contains the cause of entry in the kernel (if IRQ, syscall or except)



MIPS IRQ status

`0000` → IRQ; `1000` → syscall; `other` → Exception

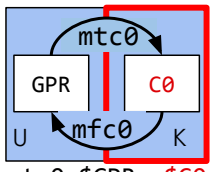
The register `c0_sr ($12)` contains the MIPS execution mode and the IRQ authorizations



Enable MIPS IRQs

User EXcept IRQ
Mode Level enable

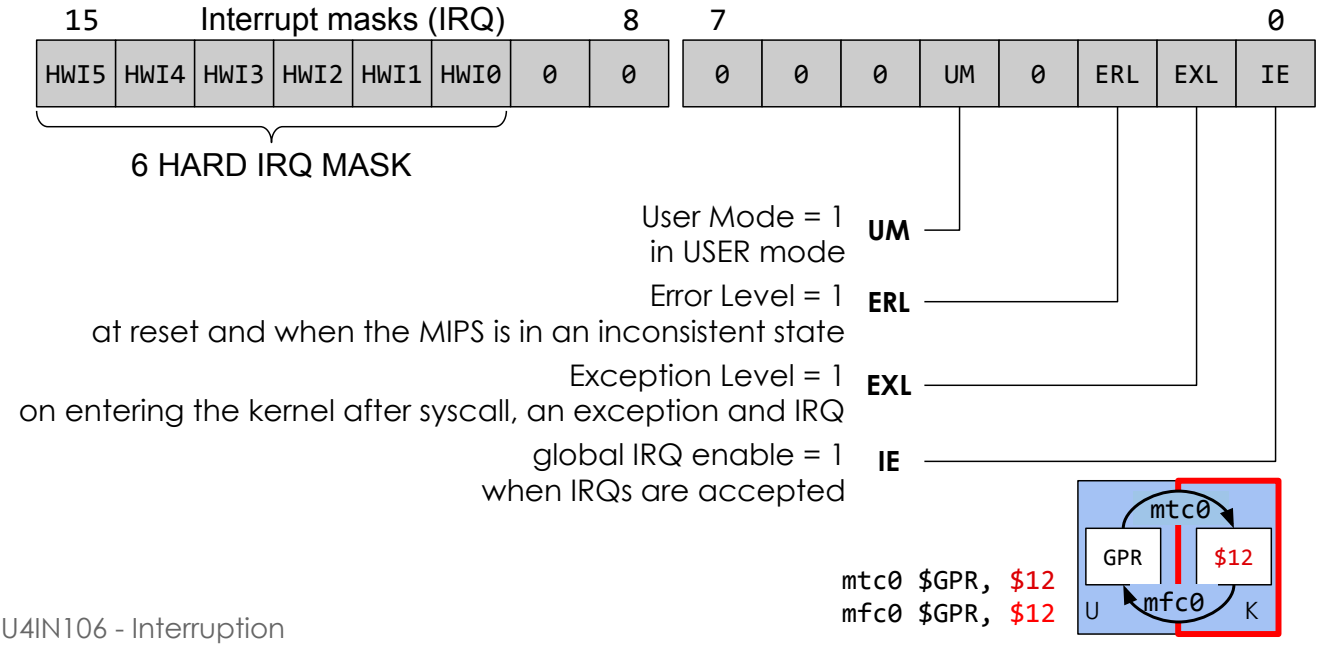
The register `c0_epc ($14)` contains the return address if it is an IRQ or the address of the current instruction for syscall and all exceptions



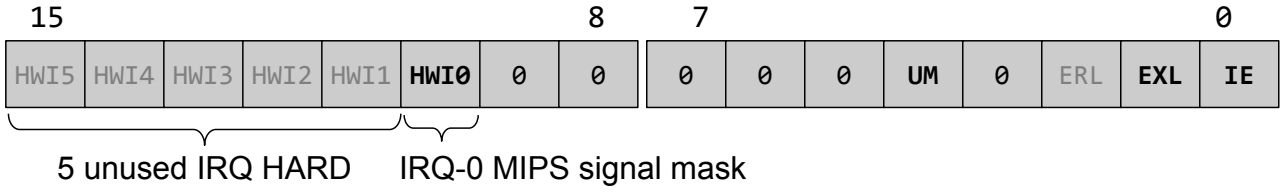

```
mtc0 $GPR, $C0
mfc0 $GPR, $C0
```

Status Register : c0_sr ($12 of c0)

c0_sr contains the interrupt line masks and the execution mode.



Behavior of the c0_sr register ($12)



Behavior of the MIPS

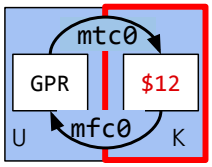
- If UM is 1: the MIPS is in USER mode
- If IE is 1: MIPS allows IRQs to interrupt the current program

EXCEPT if the ERL or EXL bits are set to 1, in effect

- **If one of the ERL or EXL bits is set to 1 then the MIPS is in KERNEL mode with IRQ masked ∀ the state of UM and IE**

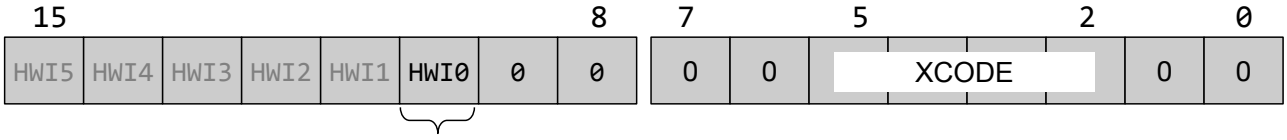
Typical values of c0_sr for the platform

- When running an application USER → 0x0411
- On entry to the kernel → 0x0413
- During execution of a syscall → 0x0401



Cause Register : c0_cause ($13 of c0)

The CR register contains the cause of entry in the kernel (after syscall, except or irq)

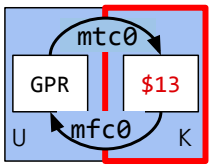


IRQ-0 signal status at the MIPS input

XCODE values actually used in this version of MIPS

- 0_{10} = 0000_2 : INT Interruption
- 4_{10} = 0100_2 : ADEL Illegal address for reading
- 5_{10} = 0101_2 : ADES Illegal address for writing
- 6_{10} = 0110_2 : IBE Bus error on instruction access
- 7_{10} = 0111_2 : DBE Bus error on data access
- 8_{10} = 1000_2 : SYS System call (SYSCALL)
- 9_{10} = 1001_2 : BP Stop point (BREAK)
- 10_{10} = 1010_2 : RI Codop illegal
- 11_{10} = 1011_2 : CPU Coprocessor inaccessible
- 12_{10} = 1100_2 : OVF Arithmetic overflow

```
mtc0 $GPR, $13
mfc0 $GPR, $13
```



Input and output of the kernel

syscall or **exception** or **interrupt**

c0_sr.EXL ← 1
: setting of the EXL bit of the Status Register, thus switching to kernel mode masked interrupts

c0_cause.XCODE ← cause number
: for example 8 if the cause is syscall

EPC ← PC or PC+4
: PC address of the current instruction for syscall and exception
: PC+4 next address for interrupt

PC ← 0x80000180
: This is where the kentry is located [syscall, exception, interrupt]

eret

c0_sr.EXL ← 0
: setting of the EXL bit of the register Status Register, thus switching to c0_sr.UM mode and with interrupt or not according to c0_sr.IE
: c0_sr.UM = 1 ⇒ user mode
: c0_sr.IE = 1 ⇒ int authorized

PC ← EPC
: designates the address of the next instruction to be executed

The coprocessor 0 (c0) registers (called system registers) are in red

Multiprogramming

- Multiprogramming is the ability of a computer to run multiple independent applications at the same time
  ⟶ one or more applications per processor.
- In the absence of virtual memory, it is not simply possible for applications to be independent because they all share the same address space.
- The GIET gives an illusion of independence, the applications cannot communicate with each other (software limitation).
  - each application has its own `main()`
  - each application has its own execution stack
  - but all these applications are in the same executable
- We use a GCC trick to know the addresses of `main()`

Reset code 1/2

- It is the reset that does everything here... the GIET is a "*static operating system*" where everything is created at startup
- All MIPS start on the same reset code
- Configuration by each MIPS
  - routing of the interrupts that it manages by configuring its ICU
  - initialization of the interrupt vector
  - configuration of the timer period
  - initialization of *its* stack *pointer*(s)
  - *initialization of thread contexts (not yet)*
- Each MIPS jumps into its own `main()`
- The addresses of `main()` are placed at the beginning of the `.data` section

Reset code 2/2

```
        .section .reset,"ax",@progbits

        .extern seg_stack_base
        .extern seg_data_base

        .func   reset
        .type   reset, %function
reset:
        .set noreorder

# get the processor id
        mfc0    $27,    $15,    1               # get the proc_id
        andi    $27,    $27,    0x1             # no more than 2 processors
        bne     $27,    $0,     proc1

proc0:
        # initialises interrupt vector entries for PROC[0]

        # initializes the ICU[0] MASK register

        # initializes TIMER[0] PERIOD and RUNNING registers

        # initializes stack pointer for PROC[0]
        la      $29,    seg_stack_base
        li      $27,    0x10000                 # stack size = 64K
        addu    $29,    $29,    $27             # $29 <= seg_stack_base + 64K

        # initializes SR register for PROC[0]
        li      $26,    0x0000FF13
        mtc0    $26,    $12                     # SR <= 0x0000FF13

        # jump to main in user mode: main[0]
        la      $26,    seg_data_base
        lw      $26,    0($26)                  # $26 <= main[0]
        mtc0    $26,    $14                     # write it in EPC register
        eret

proc1:
        # initialises interrupt vector entries for PROC[1]
```

- Note the declaration of the section
- Note the reading of the MIPS number
- For each IRQ managed by the MIPS n°0
  - Initialize the boxes of the interrupt vector with the right ISR
  - Initialize the mask of ICU #0 to let the good ISRs through
  - Initialize the TIMER n°0 period
- Note that the stack is in the stack segment
- Note the value initialized in C0_SR
- Notice how reset jumps into main [0].

```
proc1:
        # initialises interrupt vector entries for PROC[1]

        # initializes the ICU[1] MASK register

        # initializes TIMER[1] PERIOD and RUNNING registers

        # initializes stack pointer for PROC[1]

        # initializes SR register for PROC[1]

        # jump to main in user mode: main[1]

        .set reorder
        .endfunc
        .size   reset, .-reset
```


main() addresses

app.ld :

```
INCLUDE seg.ld
SECTIONS
{
    . = seg_code_base;
    seg_code:
    {
        *(.mycode)
        *(.text)
    }
    . = seg_data_base;
    seg_data:
    {
        *(.ctors)
        *(.rodata)
        *(.rodata.*)
        *(.data)
        *(.lit8)
        *(.lit4)
        *(.sdata)
        *(.bss)
        *(COMMON)
        *(.sbss)
        *(.scommon)
    }
}
```

main_pgcd.c :

```
__attribute__ ((constructor)) void main_pgcd()
{
    int opx;
    int opy;
    tty_printf(" Interactive PGCD \n");
    [...]
}
```

main_prime.c :

```
__attribute__ ((constructor)) void main_prime()
{
    unsigned prime[MAX];
    unsigned tested_value = 2;
    unsigned next_empty_slot = 0;
    unsigned is_prime;
    unsigned i;

    tty_printf("*** Starting Prime Computation ***");
    [...]
}
```

Access to device registers

- When the OS reads a device register, it is to know if a command is finished or to read data which has just came.
- The goal of GCC is to reduce the number of memory reads, when it can, it assigns registers to variables and thus avoids reads.
- This behavior is a problem for variables that are device register addresses.

  `int *registre_de_periph = address ;` → we will see the code :-)
  `while ( *periph_register == 0 );`
  → the compiler will probably assign a register of the MIPS

  Then ⟶ **`volatile`** `int *periph_register = address ;`

interrupt manager 1/2

- The GIET is invoked when an IRQ is raised
  - The GIET analyzes the `XCODE` field of the cause register `C0_CAUSE`
  - If it is an interrupt, it jumps to the `_int_handler`
  - GIET saves temporary and `EPC` registers
  - The GIET jumps to the `_int_demux` function which will invoke the correct `ISR` as it is a C function, it saves the persistent registers it uses.
  - when _int_demux returns, the GIET restores the temporary registers
  - The GIET returns to the `EPC` address
- All this code is executed in non-stop mode
- The GIET uses the user application stack

interrupt manager 2/2

- The `_int_demux` function invokes the correct `ISR`
  - It reads the ICU component to know the priority IRQ from 0 to 32
  - It uses the number as an index to access the *interrupt vector* which contains the addresses of the ISR functions
- The interrupt vector is an array of function pointers
  - Box **`n°i`** contains the address of the `ISR` function for IRQ **`n°i`**
- An ISR must
  - read or write a data or an end of command status
  - always acquire the IRQ, the way depends on the component
  - *in general, it can start a new command, not here*

Synchronous communication

- It is a communication in which the OS reads directly the registers of the registers of the device to know if there is a data to read or if it can send a command.
- tty_getc is blocking the application

```
unsigned int tty_getc(char *byte)
{
    unsigned int ret = 0;
    while (ret == 0)
    {
        ret = sys_call(SYSCALL_TTY_READ,
                (unsigned int)byte,
                1,
                0,0);
    }
    return 0;
}
```

GIET

```
unsigned int _tty_read(char *buffer, unsigned int length)
{
    volatile unsigned int *tty_address;

    unsigned int proc_id;
    unsigned int task_id;
    unsigned int tty_id;

    proc_id = _procid();
    task_id = _current_task_array[proc_id];
    tty_id  = _task_context_array[(proc_id*NB_MAXTASKS + task_id)*64 + 34];
    if(tty_id == 0)  tty_id = proc_id*NB_MAXTASKS + task_id;
    else             tty_id = tty_id - 0x80000000;

    tty_address = (unsigned int*)&seg_tty_base + tty_id*TTY_SPAN;

    if ((tty_address[TTY_STATUS] & 0x1) != 0x1) return 0;

    *buffer = (char)tty_address[TTY_READ];
    return 1;
}
```

Asynchronous communication 1/2

- A buffer in the OS records the received data when it arrives
- This buffer is read by the application (via the GIET) when it needs it
- Sequence of asynchronous steps
  - On the one hand
    - the user presses a key on the keyboard ⟶ IRQ of TTY n°i
    - the OS run the IRQ handler ⟶ run the ISR of the TTY
    - the `ISR` reads the character and places it in a buffer for the TTY n°i
  - On the other hand
    - the application asks to read the keyboard
    - executes a syscall that reads the buffer
      - returns 1 if ok
      - returns 0 if no character

Asynchronous communication 2/2

- It is a communication in which the OS reads or writes a buffer thus not the registers of the device to know if there is a data to read or if it can send a command.
- tty_getc_irq is also blocking the application

Note the declaration of the buffers

```
unsigned int tty_getc_irq(char *byte)
{
    unsigned int ret = 0;
    while (ret == 0)
    {
        ret = sys_call(SYSCALL_TTY_READ_IRQ,
                (unsigned int)byte,
                1,
                0,0);
    }
    return 0;
}
```

GIET

```
in_unckdata volatile unsigned char _tty_get_buf[NB_PROCS*NB_MAXTASKS];
in_unckdata volatile unsigned char _tty_get_full[NB_PROCS*NB_MAXTASKS] = {
    [0 ... NB_PROCS*NB_MAXTASKS-1] = 0
};

unsigned int _tty_read_irq(char *buffer, unsigned int length)
{
    unsigned int proc_id;
    unsigned int task_id;
    unsigned int tty_id;

    proc_id = _procid();
    task_id = _current_task_array[proc_id];
    tty_id  = _task_context_array[(proc_id*NB_MAXTASKS + task_id)*64 + 34];
    if(tty_id == 0)  tty_id = proc_id*NB_MAXTASKS + task_id;
    else             tty_id = tty_id - 0x80000000;

    if (_tty_get_full[tty_id] == 0) return 0;

    *buffer = _tty_get_buf[tty_id];
    _tty_get_full[tty_id] = 0;
    return 1;
}
```

IRQ processing 1/2

```
_giet:
    mfc0    $27,    $13                 /* $27 <= Cause register */
    la      $26,    _cause_vector       /* $26 <= _cause_vector */
    andi    $27,    $27,    0x3c        /* $27 <= XCODE*4 */
    addu    $26,    $26,    $27         /* $26 <= &_cause_vector[XCODE] */
    lw      $26,    ($26)               /* $26 <=  _cause_vector[XCODE] */
    jr      $26                         /* Jump indexed by XCODE */
```

```
_int_handler:
    addiu   $29,    $29,    -25*4   /* stack space reservation (19 registers to
                                       save and 4 free words to call function) */

    .set noat
    sw      $1,     4*4($29)        /* save $1 */
    .set at
    sw      $2,     5*4($29)        /* save $2 */
    sw      $3,     6*4($29)        /* save $3 */
    sw      $4,     7*4($29)        /* save $4 */
    sw      $5,     8*4($29)        /* save $5 */
    sw      $6,     9*4($29)        /* save $6 */
    sw      $7,     10*4($29)       /* save $7 */
    sw      $8,     11*4($29)       /* save $8 */
    sw      $9,     12*4($29)       /* save $9 */
    sw      $10,    13*4($29)       /* save $10 */
    sw      $11,    14*4($29)       /* save $11 */
    sw      $12,    15*4($29)       /* save $12 */
    sw      $13,    16*4($29)       /* save $13 */
    sw      $14,    17*4($29)       /* save $14 */
    sw      $15,    18*4($29)       /* save $15 */
    sw      $24,    19*4($29)       /* save $24 */
    sw      $25,    20*4($29)       /* save $25 */
    sw      $31,    21*4($29)       /* save $31 */
    mflo    $26
    sw      $26,    22*4($29)       /* save LO */
    mfhi    $26
    sw      $26,    23*4($29)       /* save HI */
    mfc0    $27,    $14
    sw      $27,    24*4($29)       /* save EPC */

    la      $26,    _int_demux
    jalr    $26                     /* jump to a C function to find the proper ISR */
```

```
restore:
    .set noat
    lw      $1,     4*4($29)        /* restore $1 */
    .set at
    lw      $2,     4*5($29)        /* restore $2 */
    lw      $3,     4*6($29)        /* restore $3 */
    lw      $4,     4*7($29)        /* restore $4 */
    lw      $5,     4*8($29)        /* restore $5 */
    lw      $6,     4*9($29)        /* restore $6 */
    lw      $7,     4*10($29)       /* restore $7 */
    lw      $8,     4*11($29)       /* restore $8 */
    lw      $9,     4*12($29)       /* restore $9 */
    lw      $10,    4*13($29)       /* restore $10 */
    lw      $11,    4*14($29)       /* restore $11 */
    lw      $12,    4*15($29)       /* restore $12 */
    lw      $13,    4*16($29)       /* restore $13 */
    lw      $14,    4*17($29)       /* restore $14 */
    lw      $15,    4*18($29)       /* restore $15 */
    lw      $24,    4*19($29)       /* restore $24 */
    lw      $25,    4*20($29)       /* restore $25 */
    lw      $31,    4*21($29)       /* restore $31 */
    lw      $26,    4*22($29)
    mtlo    $26                     /* restore LO */
    lw      $26,    4*23($29)
    mthi    $26                     /* restore HI */
    lw      $27,    4*24($29)       /* return address (EPC) */
    addiu   $29,    $29,    25*4    /* restore stack pointer */
    mtc0    $27,    $14             /* restore EPC */
    eret                            /* exit GIET */
```


IRQ processing 2/2

```
void _int_demux(void)
{
    int interrupt_index;
    _isr_func_t isr;

    /* retrieves the highest priority active interrupt index */
    if (!_icu_read(ICU_IT_VECTOR, (unsigned int*)&interrupt_index))
    {
        /* no interrupt is active */
        if (interrupt_index > 31)
            return;

        /* call the ISR corresponding to this index */
        isr = _interrupt_vector[interrupt_index];
        isr();
    }
}

void _isr_timer()
{
    volatile unsigned int *timer_address;
    unsigned int proc_id;

    proc_id = _procid();
    timer_address = (unsigned int*)&seg_timer_base + (proc_id * TIMER_SPAN);

    timer_address[TIMER_RESETIRQ] = 0; /* reset IRQ */

    _putk("\n\n!!! Interrupt timer received at cycle: ");

    char *buf = "          ";
    int date = (int)_proctime();
    _itoa_dec(date, buf);

    _putk(buf);

    _putk("\n\n");
}
```

```
_isr_func_t _interrupt_vector[32] = { [0 ... 31] = &_isr_default };
```

Initialization of the interrupt vector

Jump to the right ISR

ISR of Timer One by MIPS

ISR of TTY one per MIPS and per task

```
void _isr_tty_get_indexed(unsigned int task_id)
{
    volatile unsigned int *tty_address;
    unsigned int proc_id;

    proc_id = _procid();
    tty_address = (unsigned int*)&seg_tty_base
        + (proc_id * NB_MAXTASKS * TTY_SPAN)
        + (task_id * TTY_SPAN);

    unsigned int tty_id = _procid() * NB_MAXTASKS + task_id;

    /* save character and reset IRQ */
    _tty_get_buf[tty_id] = (unsigned char)tty_address[TTY_READ];

    /* signals character available */
    _tty_get_full[tty_id] = 1;
}

void _isr_tty_get()
{
    _isr_tty_get_indexed(0);
}
```

Conclusion

We have seen

- That the devices are accessible via memory mapped registers (in the MIPS address space).
- That the address segments have a size in power of 2 and are aligned (the address of the first byte is a multiple of the size)
- That an IRQ is a Boolean signal that allows a device to request the OS for its own account (it steals this time from applications)
- That the IRQs are routed to the MIPS through an ICU.
- That the TIMER allows the periodic lifting of IRQs
- That the TTY sends IRQs when a key is typed on the keyboard.
- That an ISR is the processing code for an IRQ.
- That the GIET initializes everything in the startup code (reset)
- Let each application start on its own main()
- That communications can be synchronous or asynchronous

In TME

- You will answer course questions to verify your understanding
- You will run several programs on the platform seen above by activating the interrupts progressively in the startup code and observe the behavior of the code.
- You will have to read the GIET code and study the execution traces to determine the time taken by the processing of the interrupt routines.