Interrupts

… An ad from our sponsors
… Now back to our show

How peripherals handled?

- To check if a device has input/output
  - Keep checking the device if the status has changed
    - Called *Programmed I/O*
    - Done via *polling*
- Is this the best way to handle peripherals?
  - No! Why?

Problem with Programmed I/O

- Requires busy-wait loop:
  - CPU can’t proceed with other computations
  - CPU can’t service other devices
    - If you are polling on one device, you can’t be polling on another
- In practice, a CPU must juggle many tasks concurrently
  - Particularly true with modern multi-tasking O/S

Interrupt Basics

- An alternative to Programmed I/O is Interrupt Driven I/O
- Basic idea
  - Have the CPU communicate with the I/O interface ONLY when “required”
  - Required means one of a few things
    - Ask I/O interface to do something (i.e. print a character)
    - I/O interface is notifying the CPU of a *‘status’* change

Interrupt Basics

- Example: When a key is pressed:
  1. Keyboard I/O interface receives scancode from keyboard device and stores it in data register
     - Local to interface - this is normal device-interface activity
  2. The I/O interface sends “interrupt request” (IRQ) **signal** to CPU
  3. In response, CPU “interrupts” current task
     - “Context” of the current task is saved

Interrupt Basics

- Example: When a key is pressed continued:
  4. CPU jumps to an “interrupt service routine” (ISR)
     - The ISR reads scancode and stores it in a queue (“buffer” – global entity)
  5. When (ISR) exits the, CPU resumes original task
     - By restoring the tasks “context”
  6. … later …
  7. When app invokes routine to read next key, it is dequeued from buffer and returned

Interrupt Basics

- Can the user “type ahead”?
- Yes, if the buffer has extra free space
- If the buffer is a single space then really no different than polling
    - Usually a multi-space buffer is used

Interrupt Basics

- What does the read routine in step 7 do if a key hasn't been pressed yet?
  - It can return immediately with a special return value
    - "non-blocking I/O"
  - It can busy wait until buffer non-empty
    - Example: on single-tasking O/S
  - It can remove app process from run queue until buffer non-empty
    - Example: on multi-tasking O/S

Interrupt - Definitions

- **Interrupt** = a signal delivered to a CPU to tell it to suspend its current task, usually just temporarily, and invoke an interrupt service routine
- **Interrupt Service Routine** = a segment of code that is specific to this interface. This routine MUST be quick, it must access the data and do any simple/minor processing and return.
- **IRQ** = **I**nterrupt **Re**q**u**est
- **ISR** = **I**nterrupt **S**ervice **R**outine
- **Context Switch** = the process of saving and restoring state (e.g. CPU state), so that multiple processes (e.g. application vs. ISR) can share a resource (e.g. CPU).

Interrupt Basics

- Is a running program being continuously executed by the CPU?
  - No!! This is an illusion presented by the computer
  - In reality, your code is being continually interrupted …
    - By I/O interfaces
    - By timers
    - By other processes in the run-queue – how?
      - Only on multi-processing O/S
    - etc.

Interrupt Basics

- How are the above actually being done?
  - Multi-tasking
    - Not the same as multi-processing
  - Two types of multi-tasking
    1. Co-operative Process Share
       - Invoking an OS System call causes a process shift
    2. Pre-emptive Processes
       - Change the active process due to a timer

Interrupt Basics

- On a very simple computer with interrupts:
  - The processor needs an IRQ input
    - The signal might be asserted by an I/O interface, such as a keyboard controller.
  - It also needs a 1-bit “IRQ mask” register which defaults to 1 on reset
    - Inside the CU, the IRQ signal is only propagated if the mask is 0.
  - The mask is a bit or bits in the CPU that determines whether or not interrupt requests will be handled NOW or not

Interrupts Basics

- The mask is use to temporarily block a request
  - If the mask is set the request is blocked
    - Once the mask is cleared the request will be processed
    - After the execution of the next instruction
- The IRQ and the IRQ mask are NOT the same!!!

Fetch-Execute Cycle

- In 2655 the fetch execute was discussed, at the basic level it runs as follows:

```
while !done
    IR ← Mem[PC]
    Increment PC
    Execute Instruction in the IR
```

- Remember for the 68000 the first two lines can be done more than once if there are extension words involved!!

Fetch-Execute Cycle with Interrupts

- How does the fetch-execute cycle change when interrupts are involved?
  - At the end of each fetch-execute (FE) cycle, the CU checks if IRQ is asserted
    - If IRQ is not asserted, the next FE cycle is started
    - If IRQ is asserted & IRQ mask is set
      - What is happening here?
    - If IRQ is asserted & IRQ mask is not set
      - The context get switched to handle the IRQ

Context Switch for Handling IRQ

- To start processing an IRQ
  1. Setting the IRQ mask
  2. Pushing PC on the stack
  3. Pushing SR on the stack
  4. Loading PC with the ISR start address
- Once the above steps have happened the fetch-execute will continue
  - The next instruction executed will be **first** instruction of the ISR!

Context Switch When Finished Handling IRQ

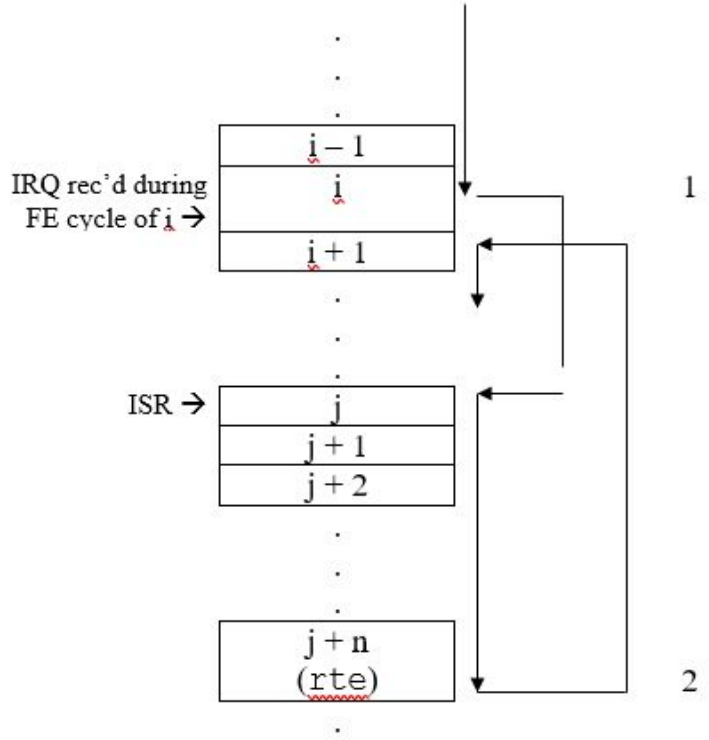
- At the end of the ISR
  1. SR must be popped and restored
  2. PC must be popped and restored
  3. The IRQ mask must be cleared
- **Note:** The steps after the ISR are in reverse order to the steps for handling the ISR
- Once the above steps have happened the fetch-execute will continue
  - The next instruction executed will be **next** instruction of another process

ISRs and the 68000

- The ISR cannot end with an ordinary “return” (i.e `rts`) instruction
  - Why?
  - Because the initial portion of the stackframe is different than a regular function call stackframe – the SR has been saved and needs to be restored. Not done by `rts`!
  - Instead, it must end with a special “return from ISR” instruction!
  - This is an `rte` instruction.

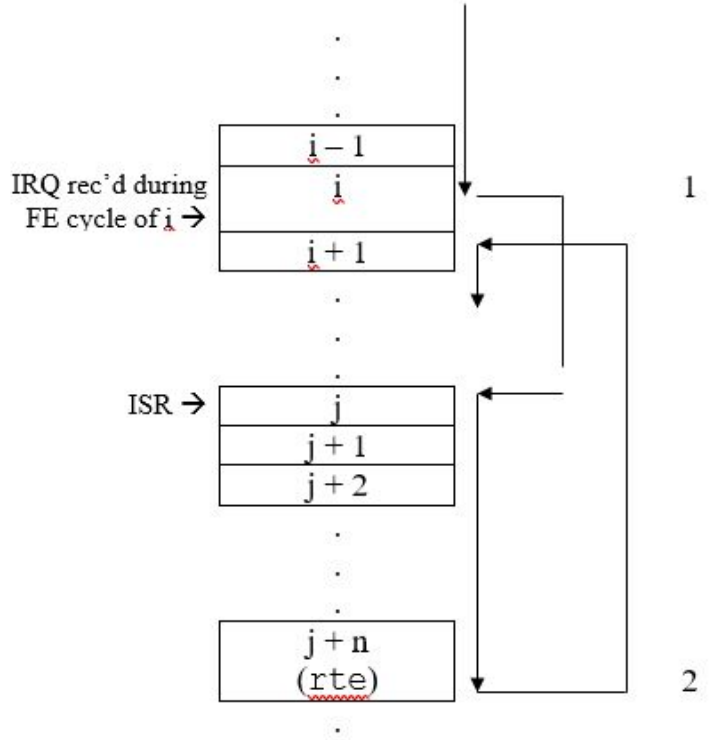
ISR Invocations

- The diagram shows how an ISR invocation progresses
- **time 1** (after i completes): IRQ mask set, PC and SR pushed, PC loaded with ISR address (vector)
- **time 2**: IRQ mask cleared, SR and PC popped



ISR Invocations

- Wait! This looks a lot like a subroutine call. What are the differences?
- Interrupted code often unaware of interrupt
  - May not even know ISR's start address
- SR must be saved/restored
  - Why?



ISR Invocations and SR

1. The executing program can be interrupted at any point in the code. So in the given code:
    - `cmp.x …`
    - `bcc somewhere`
    - And the interrupt request occurs during the cmp.x instruction
    - This instruction is completed BEFORE it is determined whether or not to handle the IRQ
    - So what happens with the `bcc` instruction if the ISR is handled?
        - The instruction relies on the CC bits determined by `cmp.x`

ISR Invocations and SR

- If the IRQ is handled then upon completion of the ISR execution resumes with the `bcc` instruction.
- The `bcc` requires the CC values from of the SR
  - Remember that ALL arithmetic/logic and **data transfer** instructions affect the CCs
  - Part of servicing the ISR will involve reading/writing the interface data registers!
    - The interface data registers are memory locations
    - Therefore the CCs **will be** altered

ISR Invocations and SR

2. The SR contains the mode bits in which the interrupted code was running
    - Need to return to the same mode.
- On a related note, why does the ISR sometimes need to save/restore other registers?
    - Because the ISR will execute instructions that will need to use registers
    - The ISR can’t know what registers are being used by the interrupted code! Why?

ISR's on the 680x0

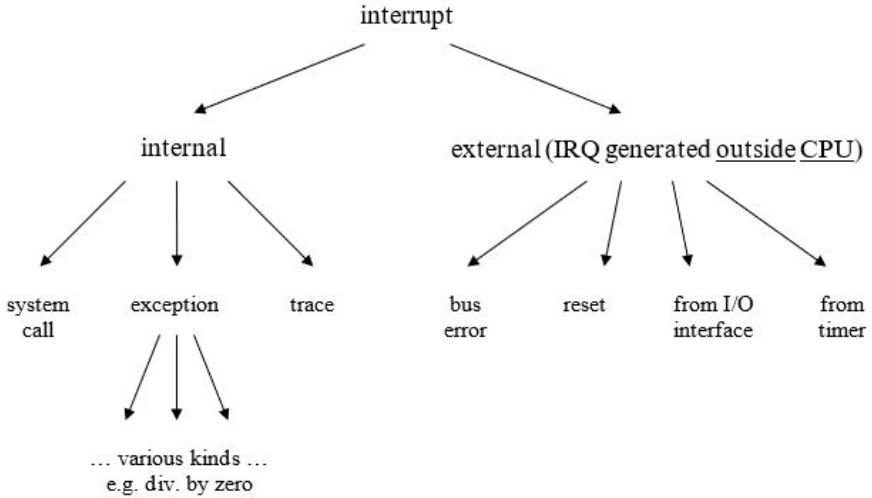
- Where is the address of the ISR held?
- Whether there is one or numerous ISRs the address(es) need to be maintained in a known location.
- This location needs to be changeable – otherwise it won't be possible to update the system.
- This location(s) is referred to as a VECTOR or Vector table.
- Determining this will be built-in to the computer system
    - i.e. CPU and additional components

ISRs and Masking

- Why do CPUs typically allow IRQs to be masked?
  - Do not want to be interrupted during system boot
    - May not have installed ISRs yet
  - May not want an ISR to be interruptible
    - IRQs are masked when going to process an interrupt request, so that the ISR being processed does NOT get interrupted by the same IRQ – the IRQ signal is NOT turned off until the data is read.
  - May not want certain “critical sections” of code to be interrupted
    - See later

Type of Interrupts

- There are several kinds of interrupt
- Unfortunately, different texts use different terminologies
- We will use the following taxonomy in this course:



68000 Documentation Terminology

- The official 68000 documentation uses different terminology

2659	68000	Some Others
Interrupt	Exception	
Internal Interrupt	‘Trap’	“s/w” or “sync.” interrupt
External Interrupt	‘Bus Error’, ‘Reset’, or ‘Interrupt’ depending on kind	“h/w” or “async.” interrupt

68000 Details

- Interrupt-handling on the 68000 is more complex than for the simple version above
    - The basic ideas are presented here, and are elaborated on in later sections
- The 68000 has a vector table which is stored in main memory
- The vector table contains 256 vectors (vectors are numbered 0-255). Each is a longword

68000 Details

- The table is always located at addresses 0-1023
  - A particular vector is located at address: vector # × 4
  - Address Error is vector #3, so its ISR start address is stored at location 12
    - As a longword … why?

Exception Vector Assignment

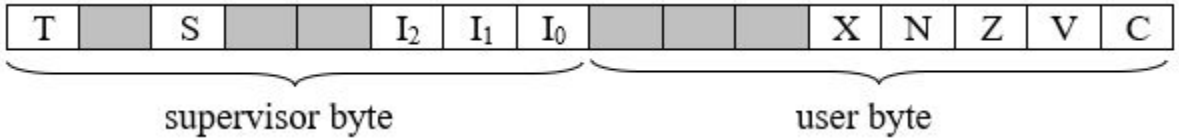
Vector Number	Address			Assignment
	Dec	Hex	Space	
0	0	000	SP	SSP after Reset
1	4	004	SP	PC after Reset
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Devide by Zero
6	24	018	SD	Chk, Chk2 Instruction
7	28	01C	SD	Trapv Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Linea (1010) Emulator
11	44	02C	SD	Linef (1111) Emulator
12	48	030	SD	(Unassigned, Reserved)
13	52	034	SD	Coprocessor Protocol Violation
14	56	038	SD	Format Error
15	60	03C	SD	DSP Transfer Interrupt
16-23	64-95	040-05F	SD	(Unassigned, Reserved)
24	96	060	SD	Spurious Interrupt
25	100	064	SD	Level 1 Interrupt -
26	104	068	SD	Level 2 Interrupt - HBL
27	108	06C	SD	Level 3 Interrupt -
28	112	070	SD	Level 4 Interrupt - VBL
29	116	074	SD	Level 5 Interrupt - SCC
30	120	078	SD	Level 6 Interrupt - MFP
31	124	07C	SD	Level 7 Interrupt -
32-47	128-191	080-0BF	SD	Trap Instruction Vectors
48-63	192-255	0C0-0FF	SD	(Unassigned, Reserved)
64-255	256-1023	100-3FF	SD	(Unassigned, Reserved)

SP = Supervisor Program Space
SD = Supervisor Data Space

68000 Details

- The table is (mostly) RAM, so vectors can be changed. On most systems (including the ST), the addresses are protected
    - Code must be run in supervisor mode in order to access the table
- When TOS boots, one of its first tasks is to initialize the vector table with various ISR vectors.
    - Many of the ISRs simply print bombs (# bombs = vector #) and unload the currently executing user program!

Structure of the SR



- T = trace bit – see below
- S = supervisor bit (0 = user mode, 1 = supervisor mode)
- I2-I0 = interrupt mask – see below
- The supervisor byte is only accessible in supervisor mode.

Interrupt Condition Detected

- The following steps are all done in hardware:
  1. A copy of SR is saved in a temp. register
  2. The SR's S bit is set, and its T bit is cleared
     - i.e. supervisor mode is entered, and trace mode is disabled
  3. PC is pushed on the system stack as a longword
  4. The original SR (from temp.) is pushed on the system stack as a word
  5. PC is loaded with the appropriate ISR vector from the vector table
  6. First instruction of ISR executed on next Fetch-Execute Cycle

Interrupt Condition Detected

- Note: the current instruction is typically completed, except for cases like reset, bus error and address error
  - For bus and address error extra data is pushed on the system stack
- Why are instructions with these types of errors not completed?
  - The error will be caused by an operand
  - i.e. a bad operand, so how could you complete the instruction
  - Example: `add.w 10,d0  ; in user mode`

Interrupt Condition Handling Finished

- What needs to happen at the end of the ISR started in step 6
  1. SR is popped and restored
     - This includes the original S and T bit values
  2. PC is popped and restored
- How is supervisor mode entered and exited?
  - Entered step 2 in enter process
  - Exited step 1 in exit process

Supervisory Mode on Startup

- What is the default for supervisory mode upon CPU reset? (On/Off)
  - **On** - CPU reset the CPU must be in supervisory mode. Why?
    - Otherwise the CPU could NEVER get into supervisor mode

Internal Interrupts

- (In this course) internal interrupt = an interrupt whose interrupt condition is detected (generated) by the CPU itself, **not** by an external IRQ signal
- An internal interrupt is usually triggered by the running program
  - On purpose (e.g. a system call)
  - Result of an exception (e.g. div. by zero)
  - A trace - is a little different

System Call

- Apps need to invoke O/S routines, but ...
  - Should not and do not need to know the routine's starting address
  - Plus an O/S routine typically needs to run in supervisor mode
- System calls are normally implemented as internal interrupts
  - Using a special instruction, the app voluntarily interrupts itself
  - The ISR is the system call handler!
    - i.e. the subroutine which implements the system calls!

System Call

- The 68000 provides the trap instruction
  - For example when TOS boots, it loads the vector table
  - This includes vectors for:
    - trap #1 (GEMDOS)
    - trap #13 (BIOS)
    - trap #14 (XBIOS)
  - When the system call handler takes over, it looks at the stack to determine which system call to perform, and to access any additional parameters. Results are returned in registers. Why?

Internal Interrupts on the 68000

- Address error:
  - Trapped if attempting to access a word/longword at an odd address
- `trapv, chk, divu, divs`:
  - These instructions generate exceptions (run-time errors) under certain conditions
  - `trapv` traps on a signed overflow

```
add.w   d0,d2
trapv           ; trap is SR's V bit is set
```

Internal Interrupts on the 68000

- `chk` - compares a source operand against the contents of a data register (the bound), and traps if the first operand isn't within range

```
chk x,d0    ; trap if x < 0 or x > contents of d0
```

- `divs, divu` - trap on division by zero

- Privilege violation:
  - Trap if attempt made to execute privileged instruction in user mode

Internal Interrupts on the 68000

- Illegal instruction:
  - Same idea, but for illegal opcodes
  - Can be used by debuggers to implement breakpoints
- Trace:
  - If trace bit is set, CPU traps to trace ISR after every instruction
  - Example: on the 68000, this is why T is cleared automatically when acknowledging an ISR
  - Used by debuggers to implement single-step

Internal Interrupts on the 68000

- Unimplemented instruction:
  - Same idea, but for “reserved” opcodes
  - Can be used to simulate unimplemented instructions in software, or to invoke a coprocessor
  - Example: Used to implement Line-A functions! The 68000 reserves opcodes of the form $Axxx and $Fxxx

    `linea0();` → C call to asm routine which contains opcode `$A000`

S & T Bits of SR

- How is the S bit set when invoking the Super call?
  - The Super call sets the S bit in the caller's SR on the stack
- How is the T bit set on a ctrl-z in the debugger?
  - The debugger is running two processes
    1. The debugger
    2. The program being debugged

S & T Bits of SR

- The debugger is active when the ctrl-z is invoked
  - A context switch will be done
    - i.e. transferring control from the debugger to the program being debugged
  - Therefore, set the T bit in the SR of the program to be debugged.
- With the T bit set the CPU will execute a single instruction and then interrupt to the Trace ISR
- The Trace ISR will perform a context switch that transfers control back to the debugger

PC and SR

- How to get the loading of the PC and SR?
  - If the PC is loaded first then will begin executing at that location and won't get the SR loaded.
  - If the SR is loaded first then loading the PC will change the SR value!
  - These need to be simultaneously done! But HOW?
  - RTE!!

External Interrupts

- (In this course) external interrupt = an interrupt which is requested via an externally delivered signal.
- Bus errors have been discussed earlier in the course. On the Atari who generates this signal, and when? What is the default response under TOS?
    - The GLUE – an external controller
        - When an access to an invalid or protected address is requested
        - Terminate the process.
- Reset is obvious

External Interrupts

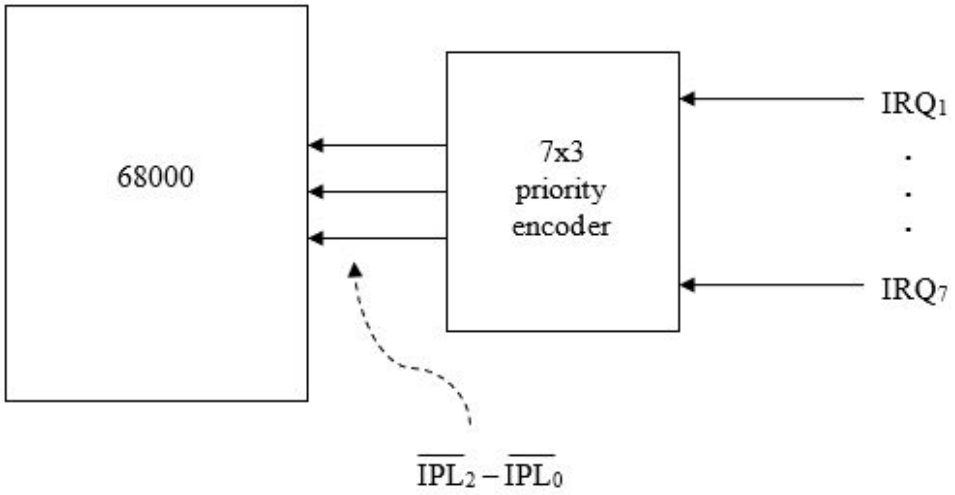
- The remaining external sources of IRQs are timers (see below) and I/O interfaces.
- E.g. a 6850 ACIA has a control register bit for enabling Rx interrupts
    - When enabled, the 6850 asserts its IRQ output whenever the “Rx full” bit in its status register is active.
- This IRQ is (indirectly) delivered to the CPU.
- Some CPUs have a single IRQ input.
- The rest of this section gives some details on how the 68000 is notified of a hardware interrupt ...

External Interrupts

- Instead of one IRQ input, the 68000 has three

Note:

- Three lines can be implemented
  - Individual levels, i.e. three distinct levels
  - As a group, i.e. a binary #
  - The 68000 uses them as a group.



External Interrupts

- Incoming interrupt requests are assigned a priority of 1 through 7
- Some CPUs have a single control bit for enabling/disabling all interrupts. Instead, the 68000 has a 3-bit interrupt mask (in SR)

IPL_{2-0}	priority	
000	7	NMI (non-maskable interrupt)
001	6	
010	5	
011	4	
100	3	
101	2	
110	1	
111	0	(no interrupt)

External Interrupts

- Note: mask means “ignore until unmasked”.
- Priorities ≤ $I_2 I_1 I_0$ are masked.
  - E.g. if mask = 0 → all interrupts allowed
  - if mask = 4 → interrupts with priority ≤ 4 are masked
  - if mask = 6 → interrupts with priority ≤ 6 are masked
  - if mask = 7 → interrupts with priority ≤ 6 are masked
- Interrupt mask → the “priority of the currently executing code”
- **Note:** A priority 7 interrupt can never be masked!

External Interrupts

- The hardware interrupt mechanism is similar to that already described:

1. (current instruction is completed) if the IRQ level is > int mask then
2. A copy of SR is saved in a temp. register
3. The SR's S bit is set, and its T bit is cleared
   i.e. supervisor mode is entered, and trace mode is disabled
4. **Interrupt mask is raised to the priority of the incoming interrupt**
5. PC is pushed on the system stack as a longword
6. Original SR is pushed on the system stack as a word
7. PC is loaded with the appropriate ISR vector from the vector table

External Interrupts

- In step 7, how does the 68000 know which vector is appropriate?
  - This will be discussed later ...
- Why is step 4 performed?
  - So the ISR cannot be interrupted by itself or by a lower-priority interrupt
- Where is the int mask stored?
  - In the SR

External Interrupts

- When is the original mask restored?
  - When the ISR executes its rte instruction (the original SR is restored)
- Upon reset, the 68000's SR is loaded with `$2700`. Why?
  - It starts in supervisor mode, with trace disabled
  - All interrupts are masked, because the vector table hasn't been initialized yet

Timers

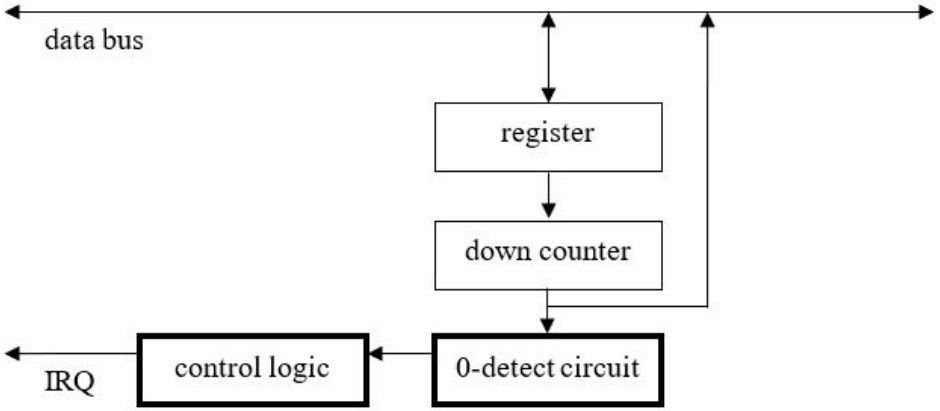
- A timer is implemented using a counter
  - Usually down-counting
- Timers are used to measure time periods, and also to trigger synchronous events.
- Timers can be “one shot”, or “free running”.
- A **one shot timer** can be used for deciding if a mouse press/release should be interpreted as a mouse click, or for automatic retransmission of a potentially lost data packet on a network.

Timers

- “Free running” means the down counter is automatically reloaded from the register at time zero
  - The timer circuit may have a control register with a mode select bit
  - Can be used for generating square wave signal in a PSG
  - A signal with period 2t can be generated if a voltage is inverted at the end of each time t
    - The larger the value in the register, the lower the frequency
  - The horizontal and vertical blank signals for controlling a monitor can be used as free running timers

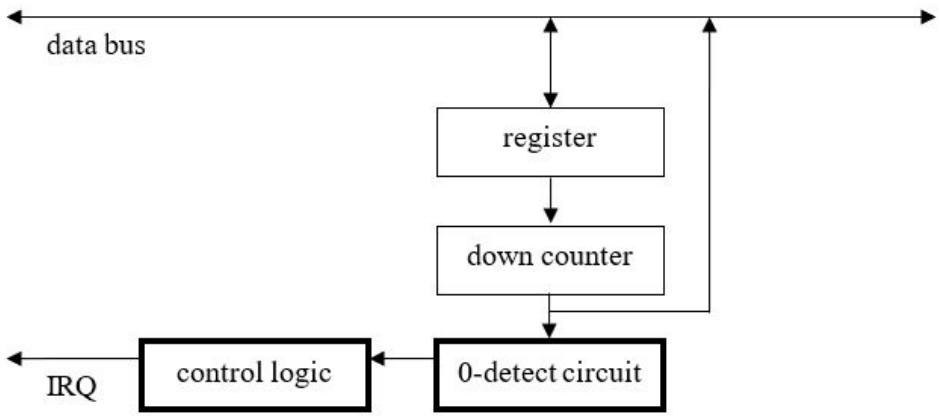
Timers

- Control and clock signals are not shown, but the basic idea is:
  - The down counter can be told to load, hold, or decrement
    - on each active edge of the clock



Timers

- Purpose of register is so the down counter can be automatically reloaded with a value, if desired, at each timeout
- The down counter can be polled, but using an IRQ is usually superior
- The register is shown as readable in this diagram, but the is not always the case

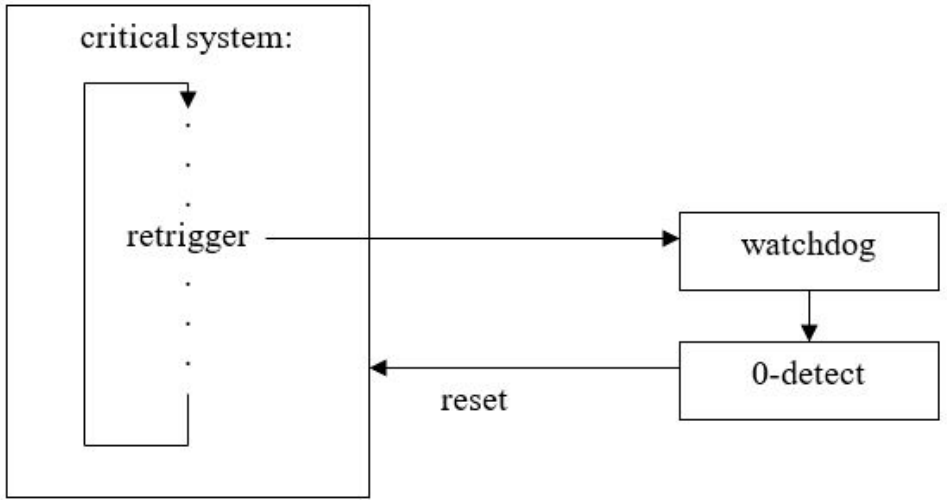


Real Time System

- A computer system with constraints on operational deadlines, such as a max. time from input event to system response (e.g. a video game is a “soft” real-time system, whereas a computer-controlled robot arm in a factory is “hard”).
  - Hardware timers must be used
  - Software timing is unreliable – why?
- A timer may be retriggerable
  - i.e. it may be possible to reinitialize it before it expires

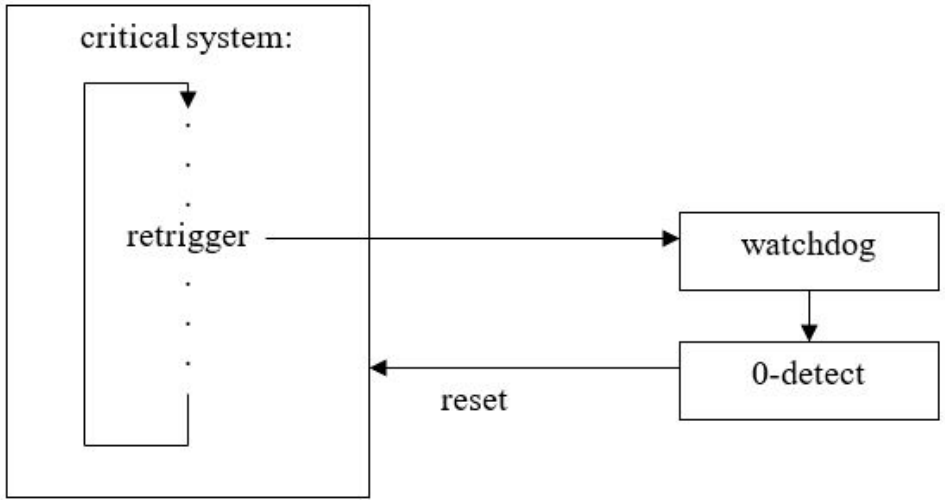
Watchdog Timer

- Switch debouncing also uses retriggerable timers
- **Idea:** when a switch contact is opened or closed, a timer is restarted



Watchdog Timer

- A press/release is officially detected only when the timer expires
    - i.e. once the switch settles into a resting state
- And only if the resting state represents a change in state.



Preemptive Multitasking

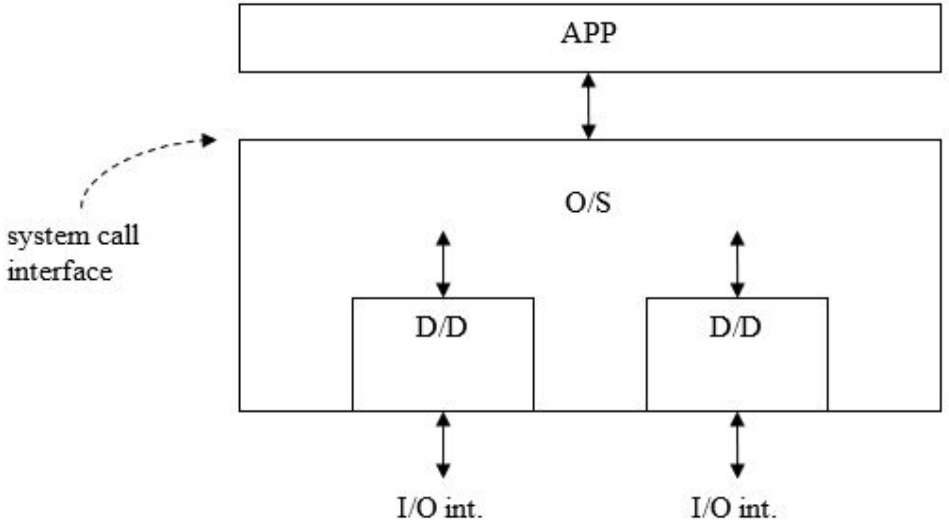
- On O/S which supports “preemptive multitasking” can set up a free-running timer to schedule process context switches
  - The “scheduler” is the timer ISR. When invoked, it returns to the next process in the run queue (not necessarily the interrupted process)!
  - So, each process in the queue gets a “time slice”

Preemptive Multitasking

- Essential scheduler operation:
  - Save context of interrupted process, including all CPU register values
    - On the 68000 this would include the process's PC and SR as found on the stack
  - Load context of next process in run queue (on the 68000 this would include altering PC and SR on the stack)
  - Return from interrupt
    - But not necessarily back to the interrupted process!

Device Drivers

- A device driver is a software module
  - Manages an I/O interface
  - Reads/writes the data from/to its registers
- Usually, device drivers are installed as O/S modules:



Device Drivers

- Note that there are three software layers:
  - Libraries are linked into the app layer
  - App issues system calls such as “read” and “write” calls to the O/S
  - O/S acts as intermediary between D/Ds and apps, and provides an abstraction layer for the app
  - Each D/D manages a particular I/O interface. Each comprises a set of routines, including at least one ISR
    - D/D modules typically conform to a standard O/S interface so that they can be “plugged in” generically

Device Drivers

- E.g. for the keyboard …
  - There is an input buffer (maybe 256 elements long). This is a circular queue with head and tail indices
  - When key pressed, driver's ISR reads input data from input reg. and enqueues in buffer at tail. Scan codes are also mapped to ASCII values
  - When app invokes "read" system call, O/S dequeues data from buffer at head and returns it to the app (as discussed earlier, the empty buffer condition must also be handled – what do `Cconin` and `Cconis` do under TOS?)

Device Drivers

- To start understanding operating systems, this is the big idea: interrupts are O/S entry points
  - After boot time, an O/S hands control off to a user processes and then sits in memory until invoked by an interrupt:
  - e.g. an internal interrupt such as a system call or an exception
  - e.g. an external interrupt due to a timer or I/O
- Each interrupt vectors to the appropriate O/S routine

Concurrency

- Now that we know about interrupts, we know that two different routines may overlap in terms of execution time
  - Even if there’s only one CPU!
- If they share resources, they must be synchronized!

Concurrency

- Consider the pseudo-code, which attempts to dequeue the latest key data from the keyboard input buffer. The buffer is accompanied by head and tail indices as well as a fill level
- Now consider the following device driver ISR code, which enqueues key data

```
if (fill > 0)      ← or busy wait until fill > 0
{
    fill = fill - 1;
    key = buffer[head];
    head = (head + 1) % BUFFER_LEN;
}
```

```
if (fill < BUFFER_LEN)
{
    tail = (tail + 1) % BUFFER_LEN;
    buffer[tail] = key;
    fill = fill + 1;
}
```

Concurrency

- Aside: % is a slow operation
  - If the buffer size is a power of 2, use &.
- Is there a disastrous time for an IRQ to be acknowledged?

```
if (fill > 0)      ← or busy wait until fill > 0
{
    fill = fill - 1;
    key = buffer[head];
    head = (head + 1) % BUFFER_LEN;
}
```

```
if (fill < BUFFER_LEN)
{
    tail = (tail + 1) % BUFFER_LEN;
    buffer[tail] = key;
    fill = fill + 1;
}
```

Concurrency

- What if it is during the execution of:

```
fill = fill - 1;
```

- … after the original fill has been fetched into a data register, but before the updated version has been stored back?
- **Critical Section** is a segment of code which accesses a shared resource
  - Critical sections must be protected, so that they have exclusive access the necessary shared resource(s)
  - **Note:** The access to the critical section does not need to be write access, but at least one access must be write access

Concurrency

- How is this accomplished?
  - In the example two slides back, the relevant interrupts should be masked at the start of the critical section and then unmasked at the end
  - The CPU can mask all IRQs, but this may be too broad
  - As we shall see, interrupts from specific sources can usually be masked individually

Concurrency

- How is this accomplished?
- The CPU can mask all IRQs, but this may be too broad. As we shall see, interrupts from specific sources can usually be masked individually
- E.g. to mask interrupts at or below a given priority on the 68000 ...

ints.h

```
/* sets IPL = mask, returns old IPL (interrupt priority level) */

int set_ipl(int mask);      /* must be called from supervisor mode */
```

ints.s

```
                xdef        _set_ipl

_set_ipl:       move.w      sr,d0
                move.w      d0,-(sp)        ; place orig. sr on stack

                lsr.w       #8,d0           ; will return orig. ipl
                and.w       #$0007,d0       ; ... in d0.w

                andi.w      #$F8FF,(sp)
                move.w      d1,-(sp)
                move.w      8(sp),d1        ; place new ipl in d1.w
                lsl.w       #8,d1
                and.w       #$0700,d1
                or.w        d1,2(sp)        ; insert it into sr on stack
                move.w      (sp)+,d1

                rte                         ; trick: when returning,
                                            ; ... install modified sr
                                            ; ... from stack!
```

Concurrency

- **Note:** invoking of Supervisor mode can be done outside the set_ipl or inside the set_ipl. It is done outside here simply because it is easier to invoke the C Super call than using the assembly language call

Sample call:

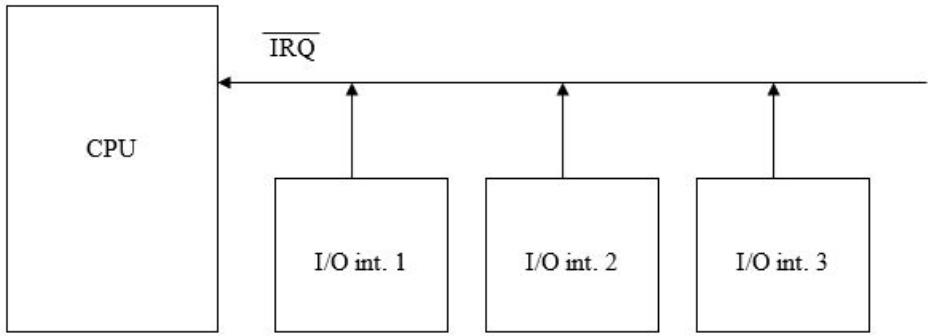
```
int orig_ipl;
int orig_ssp;

orig_ssp = Super(0);
orig_ipl = set_ipl(7);
Super(orig_ssp);

...     /* code which runs with all interrupts masked */
        /* (ST doesn't use NMI)                       */

orig_ssp = Super(0);
set_ipl(orig_ipl);
Super(orig_ssp);
```

Hardware Issues

- A computer system typically includes many IRQ sources.
- If an interrupt is requested, how does the CPU know the source?
- There are three basic techniques ...
  - S/W Polling
  - Prioritization - Auto-Vectoring
  - Vector # Transfer - H/W Polling

S/W Polling

- It is possible to construct a “normally high” bus line which can be “pulled low” by one of many units (details later)
- Here, each I/O interface has an active low IRQ output
- How many ISRs will there be?
    - One



S/W Polling

- What must it do, when invoked?
- First, note that the status register of an I/O interface typically include an IRQ flag. The ISR must poll the status of each I/O interface until it finds one who needs service, or detects a h/w error (“spurious interrupt”).
- What if multiple devices need service simultaneously? Which device is given precedence?
    - Each interface with maintain its IRQ until it is serviced – so an unserved interface will still be requesting service, just delayed
    - Service is based on the order in which the interfaces are polled, ie. “priority by position”

S/W Polling

Pros:

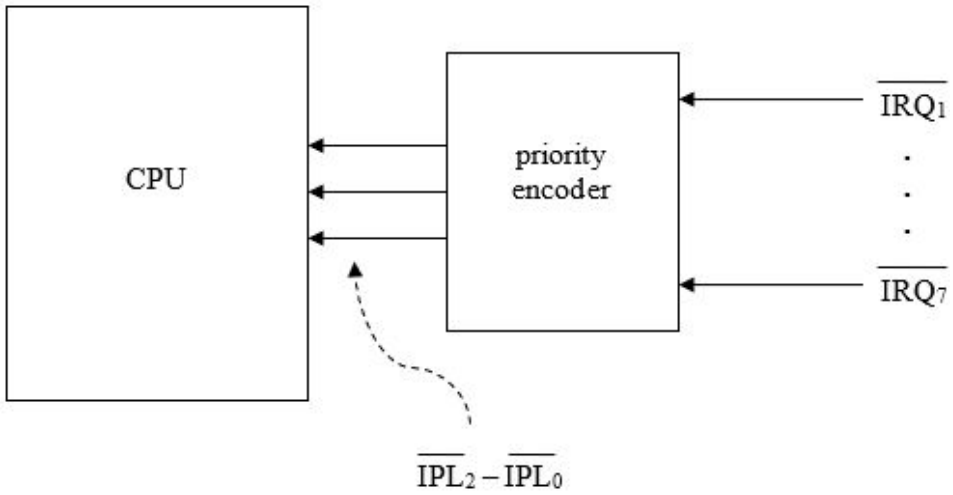
- Very simple in terms of h/w and s/w
- Easy to extend

Cons:

- Need to search for interrupt source (slow)
- Order of polling is typically set, which can lead to low priority devices being shutout

Prioritization: (Auto-Vectoring)

- Here, each interrupt source can be assigned a distinct priority, as discussed earlier:
- There are multiple ISRs (e.g. above, 7: one for each interrupt source). The priority is the vector #. During interrupt processing, interrupts of the same or lower priority are masked. Interrupts can “nest”.



Prioritization: (Auto-Vectoring)

- Requires a vector table of size 2n – 1
- Locating the ISR
  - i.e. vector, of the interrupting source is priority level * 4 + start of vector table

Prioritization: (Auto-Vectoring)

Pros:

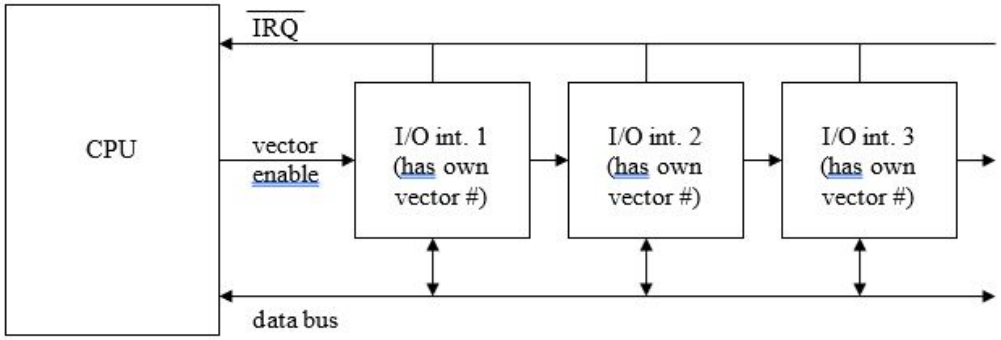
- Reduced s/w searching for interrupt source
- Nesting interrupts is possible

Cons:

- Number of priority levels is limited by h/w
- May still need to polling if more interfaces than levels, this will be done in the ISR
- Example: kbd/mouse and MIDI on the Atari

Vector # Transfer - H/W Polling

- On IRQ, the CPU activates “vector enable” (VE). The first I/O interface in the “daisy chain” either puts its vector # on the data bus, if it is asserting IRQ, or it forwards VE. (this is h/w polling)
- The CPU reads the vector # from the data bus



Vector # Transfer - H/W Polling

Pros:

- Even though searching, doing it in h/w is far faster than s/w
- Allows for many interrupting sources – based on vector size

Cons:

- Requires SMART interfaces. Each needs to determine whether to respond or pass the signal on and ability to pass the vector number.

Hardware Issues

- The 68000 uses a combination of the latter two techniques.
- When an I/O or timer interrupt request is *acknowledged*, it behaves as described earlier:

  1. (current instruction is completed)
  2. A copy of SR is saved in a temp. register
  3. The SR's S bit is set, and its T bit is cleared
     (i.e. supervisor mode is entered, and trace mode is disabled)
  4. Interrupt mask is raised to the priority of the incoming interrupt
  5. PC is pushed on the system stack as a longword
  6. Original SR is pushed on the system stack as a word
  7. PC is loaded with the appropriate ISR vector from the vector table

Hardware Issues

- … but in step 7, how does it know which vector # to use?
- It uses either (i) vector # transfer or (ii) “autovectoring” as a fall-back
- A simplified explanation of (i), when an interrupt is requested on IPL2-0:
  1. 68000 puts 111 on FC2-0 (indicates it is ack’ing an IRQ)
  2. 68000 puts priority it is ack’ing on A3-1 – why?
  3. 68000 signals a read from D7-0 using its R/W, LDS and AS signals
  4. I/O interface (or interrupt controller) puts vector # on D7-0 and asserts DTACK

Hardware Issues

- This is an example of handshaking
  2. The IPL lines can change at any time, but need to specify the appropriate interface that it is being ACKed
  4. Is answered in question 4 on the MFP lab
     - MFP base address is: `$FFFA00`
     - VR holds: `$17`
       - mc68901.pdf page 4-1 / mc68901 ds.pdf pages 5-8 (Fig 10)

Interrupt Acknowledgement

Important Lines:

AS - Address Strobe

LDS - Lower Data Strobe

DTACK - Data Acknowledgement

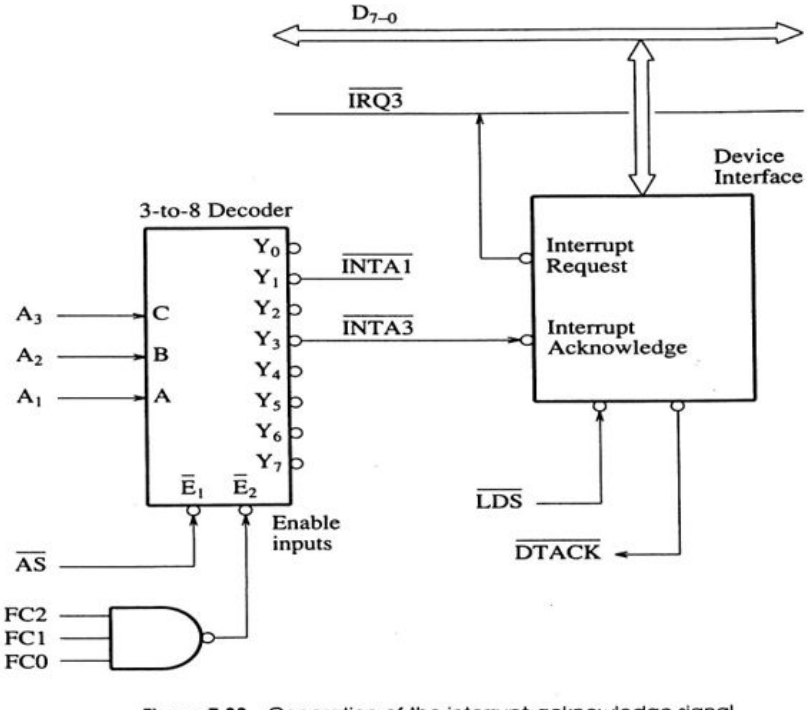


Figure 7.33 Generation of the interrupt-acknowledge signal.

Microcomputer Structures, Z.G. Vranesic & S.G. Zaky, Holt, Rinehart & Winston, 1989, page 419

Hardware Issues

- A simplified explanation of (ii):
  1. When 68000 requests vector #, a h/w unit (either the I/O interface or another chip) can respond with VPA (valid peripheral address).
  2. In this case, the 68000 aborts the read of the vector # and generates an autovector (25-31) based on the priority #
     - Autovector = table base address + 4 * (priority – 1)
- The advantage of the latter is that “dumb” I/O interfaces (which don’t know how to transfer a vector #) can still be connected

Interrupt Details

- Programmable Interrupt Controller (PIC)
  - A unit for managing IRQ sources, to offload CPU burden
    - IRQs can be programmatically prioritized, disabled, masked, etc.
- Interrupt details for the ST:
- Uses only priorities:

  6 ← IRQ from 68901 MFP (the ST's PIC)
  4 ← VBL IRQ from GLUE
  2 ← HBL IRQ from GLUE

- IPL_0 is wired to 1

Interrupt Details

- This means that level 7 (no NMI is possible) is not allowed
  - Only even numbered IRQs are possible - remember IPL signal is reverse logic
- What is the Atari's default IPL mask value? Why?
  - Default IPL mask is 3
  - Want to ignore the HBL interrupt – it happens 400 * 70 per second!

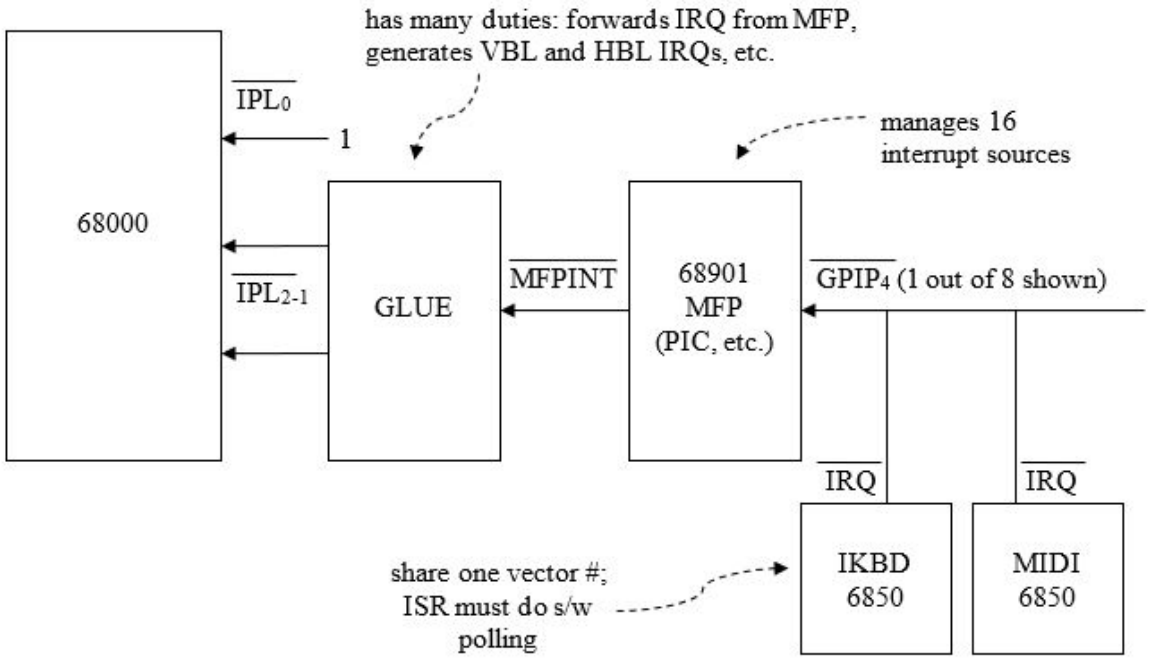
68901 MFP (Multifunction Peripheral)

- Has many duties:
  - PIC (it plays a role is managing almost all I/O interfaces)
  - Has 4 onboard timers, A-D
  - Has 1 onboard USART for the serial port
- It can manage 16 interrupt sources:
  - 8 external (on its $GPIP_{0-7}$ input pins)
  - 8 internal (timers A-D, USART)
  - It allows each to be independently disabled or masked (excessive masking of all CPU interrupts can thus be avoided)

68901 MFP

- It assigns a relative priority to each interrupt source, i.e. it prioritizes before the 68000 even sees an interrupt request.
- The diagram below summarizes the basic organization of ST interrupts
  - Among other duties, GLUE acts as an interrupt priority encoder for HBL, VBL and MFP IRQs (the former two are generated by GLUE).
  - HBL and VBL use autovectoring. When the 68000 acknowledges an interrupts and requests the vector #, GLUE asserts VPA.
  - The MFP performs the vector # transfer (e.g. the keyboard 6850 ACIA is “dumb”)

68901 MFP



The 68901 MFP will be studied further in the lab.