IRQ’s in Amber -- a short guide by Amanda

What’s supposed to happen during an ARM interrupt: [regular ARM]
– Copies cpsr into spsr_irq
– Sets appropriate cpsr bits
  • Sets mode field bits to 10010 [irq mode]
  • Disables further IRQs [nested interrupts are a no-no, but can still be interrupted by fiqs]
– Maps in appropriate banked registers
– Stores the address of “next instruction + 4” in lr_irq [r14]
– Sets pc to vector address 0x00000018
[There is a jump to an interrupt vector (at address 0x0, where each interrupt is a word, and tells the pc where to go to to handle that specific interrupt -- irq’s are at 0x18)]

To return, exception handler needs to:
– Restore cpsr from spsr_irq
– Restore pc from lr_irq
– Return to user mode

In Amber:
The interrupt handler [interrupt_controller.v] is instantiated in system [system.v]
  -There is a signal leaving this called o_irq (in system is it renamed to amber_irq)
  -This module can easily have more signals come into it to make interrupts happen
  -The rest of the module is dedicated to the wishbone. Any data necessary from the interrupt is piped over the wishbone back to the amber core.
  -If we are only using this for one kind of interrupt (the keyboard), then we can probably bypass this or just not use it, because it is a pain.
  -Important thing to know about this module: amber_irq, the signal that lets the core know that there is an interrupt, is NOT piped over the wishbone, but linked directly to the decode stage, where it is piped to the execute state and used.
  -All other signals from the interrupt controller to the core go over the wishbone
  -The interrupt controller also controls data from UART to the core, in case that is still relevant. If we are using this, using the wishbone is important for timing.

In Decode:
  -The interrupt (now called i_irq) comes into decode, because pretty much all external signals that seem useful come into the decode stage to be buffered through the pipeline until they are useful (otherwise they could be lost when the next instruction starts).
  -It is used a little in the decode logic for stuff so the execute stage is easier, but it’s nothing we really need to change (unless its not there, then it’s kind of bad, but could probably be easily worked around -- it changes something about rds and rs in the rrx instruction)
  -The decode stage creates an ‘irq_mask’ to tell the execute stage what kind of interrupt it is getting (so it knows what to do)

```
  // in order of priority !!
  // Highest
  // 1 Reset
  // 2 Data Abort (including data TLB miss)
  // 3 FIRQ
  // 4 IRQ
  // 5 Prefetch Abort (including prefetch TLB miss)
  // 6 Undefined instruction, SWI
  // Lowest
assign next_interrupt = dabt_request     ? 3'd1 :  // Data Abort
                        firq_request     ? 3'd2 :  // FIRQ
                        irq_request      ? 3'd3 :  // IRQ
                        instruction_adex ? 3'd4 :  // Address Exception
                        instruction_iabt ? 3'd5 :  // PreFetch Abort, only triggered
                                                   // if the instruction is used
                        und_request      ? 3'd6 :  // Undefined Instruction
                        swi_request      ? 3'd7 :  // SWI
                                           3'd0 ;  // none


// SWI and undefined instructions do not cause an interrupt in the decode
// stage. They only trigger interrupts if they arfe executed, so the
// interrupt is triggered if the execute condition is met in the execute stage
assign interrupt      = next_interrupt != 3'd0 &&
                        next_interrupt != 3'd7 &&  // SWI
                        next_interrupt != 3'd6 &&  // undefined interrupt
                        !conflict              ;  // Wait for conflicts to resolve before
                                                   // triggering int


// Added to use in rds_use_rs logic to break a combinational loop invloving
// the conflict signal
assign interrupt_or_conflict
                      =  next_interrupt != 3'd0 &&
                        next_interrupt != 3'd7 &&  // SWI
                        next_interrupt != 3'd6  ;  // undefined interrupt

assign interrupt_mode = next_interrupt == 3'd2 ? FIRQ :
                        next_interrupt == 3'd3 ? IRQ  :
                        next_interrupt == 3'd4 ? SVC  :
                        next_interrupt == 3'd5 ? SVC  :
                        next_interrupt == 3'd6 ? SVC  :
                        next_interrupt == 3'd7 ? SVC  :
```

```
                    next_interrupt == 3'd1 ? SVC  :
                                             USR  ;
```

In Execute:

-This is where things get ~hazy~

-So now this signal is piped into execute and is called i_status_bits_irq_mask

-In Amber, the status bits are saved with the PC (which is weird...but it saves us from having to do the ‘save cpsr into spsr_irq’ step above...), and the status_bits_irq_mask is included, so that when it returns we know how to handle it. I think pretty much all cpsr functionality is available from the pc status bits (more info about the cpsr available upon request, seriously just ask)

-Now we can use this mask to assign the signal called ‘interrupt_vector’, which is the address in the interrupt vector (that table at 0x0 I mentioned earlier) which is the address of where in the interrupt table we want to jump to, which will THEN have an address of the interrupt handler that we want. IRQ is always at 0x18, by ARM convention.

```
assign interrupt_vector = // Reset vector
                    (i_interrupt_vector_sel == 3'd0) ? 32'h00000000 :
                    // Data abort interrupt vector
                    (i_interrupt_vector_sel == 3'd1) ? 32'h00000010 :
                    // Fast interrupt vector
                    (i_interrupt_vector_sel == 3'd2) ? 32'h0000001c :
                    // Regular interrupt vector
                    (i_interrupt_vector_sel == 3'd3) ? 32'h00000018 :
                    // Prefetch abort interrupt vector
                    (i_interrupt_vector_sel == 3'd5) ? 32'h0000000c :
                    // Undefined instruction interrupt vector
                    (i_interrupt_vector_sel == 3'd6) ? 32'h00000004 :
                    // Software (SWI) interrupt vector
                    (i_interrupt_vector_sel == 3'd7) ? 32'h00000008 :
                    // Default is the address exception interrupt
                                                       32'h00000014 ;
```

-Then o_daddress_nxt is decided. daddress_sel is a signal piped in from decode, and the mask that is used to decide daddress_nxt is from there.

-pc_nxt is decided similarly, (in this case it is interrupt_vector calculated above)

-In the section titled “Register Update”, an interrupt is grounds for saving the whole pc, because both that return address and those status bits are needed. And the new status bits and pc need to be set so the IRQ mode can run properly - (that status_bits_irq_mask is also saved separately in the register file)

-This section is kind of sketchy; its hard to tell what’s going on exactly with all these signals and where the old ones are being stored

It Seems to me that none of the registers are banked here in the transition. I think that, in whatever irq handler that ends up getting written, there needs to be the saving of registers before continuing execution (usually this happens on the stack, but another data structure can be substituted easily), so that the user mode registers don’t get overwritten with assembly data; that is Bad News Bears. Of course, upon return we, as the user, need to unbank those user mode registers, restore the status bits, and restore PC back to its old mode before returning. I am a little at a loss of how to do that, though, when CPSR is no longer a recognizable entity (that was the REASON it was a separate entity).