Better Devices

• Now SW, KEY can be read
  – Problem: several instructions needed to detect change
• A better input device e.g. SW
  – Have the usual 10 data bits (what we have now)
  – Add status bits
    • A “ready” bit to tell us if there has been a change in data
      – Set to 1 automatically when any SW changes value
      – Cleared to 0 when data is read
    • An “overrun” bit to tell us if we missed something
      – Set to 1 if SW changes value and Ready is already 1
      – Cleared to 0
  – Add control bits, such as Interrupt Enable (IE)
    • If 1, Ready=1 causes interrupt request to processor
    • We’ll have to figure out interrupts later, for now this bit has to be 0 😊
Connecting Memory-Mapped Devices

• We have a few devices
  – Each with several memory-mapped registers (data, control, status, etc.)
• All connected together using lots of if-statements
  – Hard to make changes w/o creating bugs
  – Hard to debug if there is a problem
• Real systems can have 100s of devices
  – Each with control, status, etc. registers
  – The “if(dmemaddr==0xFFF0)…” approach is messy
• We need a better way of connecting devices
The Bus

- We read or we write at any given time
  - Address, to know what is the location
  - Data (either to or from memory/device)
  - WE (says if we are reading or writing)
- There are multiple entities attached
  - Processor itself (it reads/writes data)
  - Memory
  - Key, sw, display, etc.
- What goes on the address and WE wires?
  - Easy – the processor always initiates an operation
  - We say that the processor is the “bus master”
  - It drives the address and WE signals, others just listen
Another One Drives The Bus

• But data wires are different
  – If processor is writing, it drives the data lines
  – If processor is reading, memory/device drives data

• When does the processor drive the data bus?
  WrMem?DataToMem:Zs

• When does memory drive the data bus?
  – MemEnable is our “address is in the first 8kB” signal:
    (MemEnable&&!WrMem)?...

• When does a device drive the data bus?
  ((memaddr_M==MyAddress)&&!WrMem)?...
Putting it all together

wire [(DBITS-1):0] abus;
tri [(DBITS-1):0] dbus;
wire we;

// In the processor
assign abus=memaddr_M;
assign we=wrmem_M;
assign dbus=wrmem_M?wmemval_M:{DBITS{1'bz}};

// Attach some sort of a device
Timer #(.BITS(DBITS), .BASE(16'hFFFFFF20), ...) timer(
   .ABUS(abus), .DBUS(dbus), .WE(we),
   .INTR(intr_timer),
   .CLK(clk), .LOCK(lock), .INIT(init),
   .DEBUG()
);

CS 3220 Fall 2011 - Prof. Milos Prvulovic
module Timer(ABUS, DBUS, WE, INTR, CLK, LOCK, INIT, DEBUG);
  parameter BITS;
  parameter BASE;
...
  input wire [(BITS-1):0] ABUS;
  inout wire [(BITS-1):0] DBUS;
  input wire WE, CLK, LOCK, INIT;
  output wire INTR;
...
  wire selCtl=(ABUS==BASE);
  wire wrCtl=WE&&selCtl;
  wire rdCtl=(!WE)&&selCtl;
...
  assign DBUS=rdCtl?{...Contents of the TCTL register...}:
    rdCnt?{...Contents of the TCNT register...}:
      {BITS{1'bz}};
Practical issues

• Must make sure only one entity drives the bus
  – In our code, this is true at the end of each cycle
  – But what happens during a cycle?
• Our bus wires is really just OR gates
  assign DBUS=rdCtl?...:{BITS{1'bz}}; // In timer
  assign DBUS=rdKey?...:{BITS{1'bz}}; // In keys

  assign DBUS_t=rdCtl?...:{BITS{1'b0}}; // In timer
  assign DBUS_k=rdKey?...:{BITS{1'b0}}; // In keys
  assign DBUS= DBUS_t | DBUS_k | ...;

• Multiple drivers won’t damage our circuitry
• If values correct (one driver) at end of cycle, when we get them into FFs, it’s all OK
Where else can we use this idea?

• Wherever we combine outputs from multiple sources into a single value
• E.g. our ALU result output can be bus-like
  – Logic (and/nand/or/nor/xor) drives it if alulog==1
  – Adder/Subber drives it if aluadd==1
  – ...
  – Would make it easier to add more ALU units
    • E.g. a shifter, a multiplier, etc.

• E.g. reg value with forwarding can be bus-like
  – Drive regout if no forwarding
  – Drive aluout if forwarding ALU->RR
  – Drive memout if forwarding MEM->RR
• We will add “proper” devices for Project 4
  – The KEY and SW get control and status registers
  – The LEDR, LEDG, HEX get read/write capability
    • Lets us read what is being displayed
  – Add a new programmable timer device
    • Will let us do things that depend on real time
    • E.g. we can do something exactly every 2 seconds
• Interrupt support is optional in Project 4
  – Will be required in Project 5!
• And we’ll write a “wristwatch” application
  – Same as Project 1, but this time it’s a program for our processor!
• **KDATA register at FFFFFFF00**
  – Same as before
  – Current state of KEY[3:0]
  – Writes to these bits are ignored
• **KCTRL (control/status) register at FFFFFFF04**
  – Bit 0 is the Ready bit
    • Becomes 1 if change in KDATA state detected
    • A read from KDATA changes it to 0
    • Writes to this bit are ignored
  – Bit 2 is the Overrun bit
    • Set to 1 if Ready bit still 1 when KDATA changes
    • Writing a zero to this bit changes it to zero, writing a 1 is ignored
  – Bit 8 is the IE bit
    • If 0, KEY device does not generate interrupts
    • Can be both read and written
  – Start the device off with KCTRL all-zeros!
• SDATA register at FFFFFFF10
  – (Almost) the same as before
• SCTRL (control/status) register at FFFFFFF14
  – Exact same bits as SCTRL, but these apply to SDATA
  – Problem: if SW not debounced, will Overrun often
  – Solution: SDATA holds debounced value
• Debouncing SDATA
  – Holds debounced value of SW (not raw SW value)
  – SDATA changes only if the “raw” SW value is stable for at least 10ms
    • E.g. if SW is 10’h000, changes to 10’h001, then 1ms later back to 0, SDATA just stays at 0 the whole time and the Ready bit does not change
• Almost the same as before
  – Writes to FFFFFFF80, FFFFFFF90 and FFFFFFA0 change what is shown on HEX, LEDR, and LEDG
    • Only bits that actually exist are written
    • E.g. writing value 0xFFFFFFFF to FFFFFFA0 is the same as writing 0x000000FF (LEDG only has 8 actual bits)
  – But now reads from these addresses return what is currently shown
    • Bits that don’t exist are returned as zero, e.g. after writing FFFFFFFFF to LEDG, a read returns 0x000000FF
New Device: Timer

- **TCNT at FFFFFFF20**
  - Read returns current value of the counter
  - Write sets value of the counter
  - Incremented every 1ms

- **TLIM at FFFFFFF28**
  - Write sets the value, read gets the value
  - When TLIM is zero, it has no effect (counter just keeps counting)
  - When TLIM!=0, it acts as the limit/target value for the counter
    - If TCNT==TLIM-1 and we want to increment TCNT, we reset TCNT back to zero and set the ready bit (or overflow if Ready already set)
    - If TLIM>0, the TCNT never actually becomes equal to TLIM (wraps from TLIM-1 to 0)

- **TCTL (control/status) register at FFFFFFF24**
  - Same bits as KCTRL and SCTRL
  - “Ready” and Overflow bits set as described for TLIM
  - Writing 0 to the Ready or Overflow bit changes it to 0, but writing 1 to one (or both) of these is ignored
    - Properly written code should not write 1 to “Ready, but if it does then it has no effect

- Start the device off with TCNT, TLIM, TCTL all-zeros!
• Polling the devices to detect events is inefficient
  – Processor kept very busy, but nothing actually done
• We want to let devices interrupt the processor
  – Processor can do whatever needs doing (e.g. sorting)
  – When device needs attention, interrupt
  – Processor executes interrupt handler to take care of device-related activity
  – Processor returns back to whatever-needs-doing 😊
• Each device has an IRQ signal
  assign IRQ = Ready && IE;
• Processor’s IRQ signal = OR all device IRQs
  – Interrupt the processor if any device wants to interrupt it 😊
Interrupting the pipeline

• Monitor the IRQ signal
  – Normal execution continues if IRQ is 0
• Save address of next instruction
  – Where? Why not RA?
  – Which instruction is “next”? 
• Indicate which interrupt was raised
  – How?
• Jump to interrupt handler routine
  – Where is it?
  – How to safely divert fetching?
  – IRQ will still be 1, should not get stuck
    (forever doing the jump-to-interrupt thing)
• Saving the return address
  – Need a register for this (call it IRA)
  – We have R24..R27 reserved for system use
    • This is system use 😊 so maybe we can use one of these

• Where to jump on interrupts
  – Simplest way – some low fixed address, e.g. 0x100
    (this is below our program-start address at 0x200)
  – Fancier way – address can be programmed
    • Another register - IHA (interrupt handler address)

• How do we know which interrupt we had?
  – There could be many devices
  – Each can have several interrupt-causing events
Which interrupt?

• Option 1: Handler checks each device
  – Same handler for all interrupts
  – Handler checks Ready bits of devices in priority order

• Option 2: Different handler for each device/cause
  – Separate handler for each device, no need to check
  – Need many handler addresses
  – Some devices very similar, could use the same handler

• Option 3: Cause-ID number, pass it to handler

• We’ll use the Cause-ID approach
  – Put ID of interrupting device in a register, call it IDN (interrupt device number)
• Need some way to disable all interrupts
  – Must disable all ints before we divert fetch to handler
    • Why do we have to do this?
  – Handler can enable ints again when it it’s safe
    • When is it safe?

• Typical approach
  – Have a special control/status register
    PCS (processor control and status)
  – Some bit (e.g. bit 0) is “Interrupt Enable” bit
  – We’ll need a few more control/status bits soon
So far, we need four special registers
  – And we have R24..R27 reserved – perfect match
  – Or is it?

Must be automatically updated in HW
  – This will create some tricky problems for our pipeline
  – Regs written in last pipeline stage (let’s call it W), so on interrupt we must write R24..R27 in the last stage
  – But… that is three registers to write in one cycle!
  – We really do not want to do that!

OK, write them one by one
  – When taking interrupt, the last pre-interrupt instruction reaches the W stage
  – The three cycles after that - update the IRA, IDN, and PCS registers
  – Then we can do the first instruction of the interrupt handler

This is pretty messy
  – Need some sort of a state machine for the three cycles, etc.
Plan B – Separate special registers
- We will have 4 extra registers for IRA, IHA, IDN, PCS
- Three of them written by hardware when int taken

But... how does our program read/write these
- Using special instructions, of course 😊
- RSR Rx,Sx – Read system register Sx (into Rx)
- WSR Sx,Rx – Write system register Sx (value from Rx)
IntHand:
; Save general purpose registers to stack

RSR A2,IDN ; Get cause of interrupt
; Decide what to do depending on A2
...
...
; Restore general-purpose registers from stack
...
; Return and enable interrupts

How?
Saving Registers in Interrupt Handlers

• Interrupt handlers need to use a stack
  – Need to save *all* regs we modify
  – Need to save IRA if we want to enable ints again

• Can’t use the “normal” stack and SP for this!
  – How much space does the app need for its stack?
  – Is the app’s SP always in a usable state?

  SW   RA,-4(SP)  ADDI   SP,SP,-4
  ADDI  SP,SP,-4  ADDI  SP,SP,-4  vs  SW   RA,0(SP)
Need a separate SP

- Idea: a separate SSP register
  - This is what R24..R27 are for
  - So R24 is now SSP
IntHand:

; Save general purpose registers using SSP
ADDI SSP, SSP, -4 ; If only saving one reg
SW A0, 0(SSP)

RSR A0, IDN ; Get cause of interrupt
; Decide what to do depending on A2
...

; Restore general-purpose registers
LW A0, 0(SSP)
ADDI SSP, SSP, 4
; Return and enable interrupts
Returning from interrupt?

• What needs to be done
  – Enable interrupts
  – Jump back to interrupted program
• Let’s try this
  – Enable interrupts (write to PCS)
  – Use JAL to jump back
• Problem 1: Need a register for JAL target
  – Not a problem - can use R25, R26, or R27 for this
    • E.g. RSR R24,IRA, then JAL R24,0(R24)
• Problem 2: New interrupt can come before JAL
  – When it returns, R24 will be have the address of our JAL
    and we end up in an infinite loop (JAL to itself)
  – Need to enable-interrupts-and-return (all at once)
Return from interrupt
- Jumps to address in IRA
- Enables interrupts (set IE to 1 in PCS)

Better: Restore IE to what it was
- Will come handy later (for non-maskable interrupts)

Add OIE (old-IE) bit to PCS
- When taking interrupt, IE copied to OIE, then set to 0
- RETI now copies OIE to IE
- What about nesting interrupts?
  - Must save OIE before enabling interrupts in interrupt handler!
We can have up to 32 special regs in RSR/WSR

- We need only four of those:
  - 0: PCS (Processor Control/Status)
  - 1: IHA (Interrupt Handler address)
  - 2: IRA (Interrupt Return Address)
  - 3: IDN (Interrupt Device Number)
- 4..31: Reserved
Which address to save into IRA?

• Option 1:
  – Save pcgood from the stage that generates pcgood
  – Flush all stages before that, set PC to IHA, start fetching from there

• Option 2:
  – Start fetch from IHA, let already-fetched insts finish
    • Save pcgood when the last-fetched instruction produces it
    • This happens a few cycles after fetch from IHA started
  – Why would we want to do this?
    • No wasted cycles
  – Why would we not want to do this?
    • More complex, and interrupts are taken rarely (OK to waste a few cycles)
• Say we produce pcgood in A stage
• Say we get an int when a bubble-NOP in A
• What will be saved into IRA?
  – A bogus address produced by our bubble-NOP
  – Remember that we prevented NOP from setting mispred to 1
  – This is the same problem – keep bubble-NOPs away for your PC
• Solution 1: Delay taking the interrupt
  – Take int only if a “proper” inst in A
  – The delay is at most a few cycles
• Solution 2: Remember the last valid pcgood
  – If bubble-NOP, use pcgood of last non-NOP inst!
• Either way, we need to know if we have a bubble or not
  – Have a bit for this purpose, set to 0 in decoding logic
  – When NOP created by flush and stall, set this bit to 1
Multiple devices may want to interrupt
  – We will interrupt the processor
  – But which ID to put in IDN?

OK, so we need some sort of priority
  – We’ll use priority in order of IDs (0 is highest)

How to implement priority?
  – Option 1: Daisy chain
  – Option 2: Interrupt controller with priority
• Each device has
  – Input IRQ signal (from a lower-priority device)
  – Output IRQ signal
  – Connection to ID signal with tri-state logic
    • Drive its own ID there only if no higher-priority device has IRQ=1
    • But how do we know this?
  – OK, so we need some signal (e.g. INTA) that goes back to lower-priority devices

  – This is getting a bit too complicated for our purposes!
Interrupt controller

- Takes IRQ signal from each device
- Outputs overall IRQ signal to the processor
  - This is simply an OR of the incoming IRQ signals
- Outputs the number of the “winner” device as ID
  - Circuit that does this is called a “priority encoder”
  - Processor puts this number into IDN when int taken
• Where is the INTA signal?
  – We replaced it with software!
    • Device keeps IRQ at 1 until handler does something!
  – But now we must first remove the cause of the interrupt before we enable interrupts again!
    • For the interrupt-causing device, set Ready bit to 0 (or set IE bit to 0)
  – Otherwise, handler gets interrupted by the same interrupt cause as soon as it enables interrupts!
Format of PCS

• Which bit means what:
  – Bit 0 is IE
  – Bit 1 is OIE

• What to do on interrupt/exception
  – \{OIE,IE\} \leq \{IE,1'b0\}

• What to do on RETI
  – IE \leq OIE
Option 1: Give each a new primary opcode
  - But... they don't need immed bits!

Option 2: Hijack one of existing instructions
  - E.g. BEQ Rx,Ry,0 ordinarily makes no sense
    - So we can declare that this is our RSR Sx,Ry
  - Bad idea – breaks backward compatibility
    - What is someone used BEQ R0,R0,0 in their code
      (makes no sense, but people can do things that don’t make sense)

Option 3: Use a secondary ALUR opcode
  - But... these are not ALU instructions...

Option 4: Create a new SYS primary opcode
  - Then use secondary opcodes within that
  - This is what we will do
Format of new system instructions

- Primary opcode for SYS is 6’o70
- RETI
  \{opcode1,rx ,ry ,rz ,opcode2\}
  \{SYS ,00000,00000 ,00000,000001\}
- RSR Rx,Sx
  \{opcode1,rx ,ry ,rz ,opcode2\}
  \{SYS ,Sx ,Ry ,00000,000010\}
- WSR Sx,Rx
  \{opcode1 ,rx ,ry ,rz ,opcode2\}
  \{SYS ,Rx ,Sx ,00000, 000011\}
Where to read/write system registers?

- **Approach 1**: Same as non-system registers
  - Must take care of forwarding these values, too
  - E.g. WRS followed by RSR in a five-stage pipeline
- **Approach 2**: Like memory (R/W Sx in one stage)
  - No forwarding of Sx values!
  - Which stage should it be?
    - After we read Rx (so WSR can do Rx -> Sx)
    - Before we write Rx (so RSR can do Sx -> Rx)
    - But not in any stage that gets flushed (can’t undo writes to Sx)!
    - So a good stage for this is where we read/write memory
      (which has the exact same after-read-before-write-can’t-flush issues)
Jumping to the Interrupt Handler

• Option 1: Extend PC selection

```verilog
always @(posedge clk) if(lock) begin
  if(init)       PC<=16'h200;
  else if(intreq)     PC<=IHA;
  else if(mispred_B)  PC<=PCcorr_B;
  else if(!stall_F)   PC<=PCpred_F;
end
```

• Option 2: Treat like a branch misprediction
  – If intreq is active, IHA goes into pcgood

• Which one to use?
  – Option 1 keeps the pcgood computation simpler!
  – But Option 2 less likely to affect clock cycle time...

CS 3220 Fall 2011 - Prof. Milos Prvulovic
• Make a priority encoder
  
  wire [3:0] intnum=
      intr_timer?4'h1:
      intr_keys? 4'h2:
      intr_sws? 4'h3:
      4'hF;

• Put this in IDN when taking interrupt
When to take the interrupt?

wire intreq = (!reset) && (!isbad_A) 
  (PCS[0] && (intr_timer || intr_keys || intr_sws))

• Be careful with stages
  – Which stage writes PCS, IRA, etc.?
    • Should be the same stage for WSR and int-taking writes
    • Must be after we read normal regs (so WSR can work)
    • Must be after the inst is “safe” from flushes, otherwise we can get IRA from an inst that was not supposed to execute at all!
  – Which stage reads PCS, IRA, etc. in RSR?
    • Must be before last stage (so RSR can work)
    • Much easier if it’s the same stage where they are written (so we don’t have dependences/hazards between RSR and WSR)
Special Registers - Implementation

• As a register file
  – Good: Read/Write (RSR, WSR) similar to R* registers
  – Bad: On int, need to read/write several of these at once
    • Will result in many-ported register file (slow and expensive)

• As a bunch of named registers
  – Good: Can use PCS, IDN, etc. as needed for Rd/Wr
  – Bad: RSR, WSR require a case statement
    • Not too bad, “case(sregno)” and update each if its number is selected

• Either way, must carefully resolve conflicting r/w
  – E.g. check PCS[0] for IE while WSR writing 0 to IE
    • If intreq is 1, do we take interrupt here or not?
• For RSR, use rregno1 as src register number?
  – OK, but make sure it does not mess up forwarding!
  – If forwarding based only on rregno, will forward from ADDI R1, R2, R2 to RSR R3, IHA
  – Will need rdrreg1, rdsreg1 signals
    • To tell us which register the rregno1 refers to
Project 4

- Implement I/O devices in this new way
  - Control/Status regs, timer device, etc.
  - Interrupt support is optional in Project 4
    - For now your devices can have the IE always set to 0 (no int)
    - And you don’t have to have RSR, WSR, and RETI instructions
    - But all of this will be needed in Project 5
  - Processor must run Sorter2 in 45 seconds or less

- Write an application, Clock.a32 and Clock.mif
  - Implements the clock part (clock and clock-set mode) from Project 1 using our processor
    - It must all be implemented as assembler code for our processor, i.e. do not implement the processor as Verilog code from Project 1
    - Stopwatch mode is optional for Project 4
    - Will be required for Project 5