Lecture 5: Memory-mapped I/O review, APB, start interrupts.
Mostly APB though 😊

January 21st 2016
Reading

• Read sections 8.1 and 12.1 of LeeSeshia (see references page)
  – It’s a decent overview of processor types & scheduling
  – Don’t worry about understanding their FIR filter example
    • Just the idea why I might want specialized digital signal processors.

• The whole book is a very interesting take on embedded processors
  – Focused on modeling and simulation
Today

• Memory-mapped I/O and bus architecture review

• ARM’s APB bus in detail

• Start on interrupts
Memory-mapped I/O

• The idea is really simple
  - Instead of real memory at a given memory address, have an I/O device respond.
    • Huh?

• Example (this time word aligned)
  - Let’s say we want to have an LED turn on if we write a “1” to memory location 8.
  - Further, let’s have a button we can read (pushed or unpushed) by reading address 4.
    • If pushed, it returns a 1.
    • If not pushed, it returns a 0.
Write assembly programs.

- LED on if button pressed
- LED off if button pressed
Now write button pressed=LED on in C.

```c
#define SW ((volatile  uint32 *) 0x4)
#define LED ((volatile  uint32 *) 0x8)

main(int argc, char * argv[])
{
    while(1)
    {
        *LED=*SW;
    }
}
```

1. What is uint32?
2. Why volatile? What does that do?
3. Could I get rid of the dereference in the code? Should I?
Modern embedded systems have multiple busses

Expanded 373 focus

Historical 373 focus

Atmel SAM3U
Bus terminology

- Any given transaction have an “initiator” and “target”

- Any device capable of being an initiator is said to be a “bus master”
  - In many cases there is only one bus master (single master vs. multi-master).

- A device that can only be a target is said to be a slave device.

- Some wires might be shared among all devices while others might be point-to-point connections (generally connecting the master to each target).
Today

- Memory-mapped I/O and bus architecture review
- ARM’s APB bus in detail
- Start on interrupts
APB is a fairly simple bus designed to be easy to work with.

- Low-cost
- Low-power
- Low-complexity
- Low-bandwidth
- Non-pipelined
- Ideal for peripherals
Let’s just look at APB writes (Master writing to device) as a starting point.

- We’ll add reads shortly.
Notation

Clock

HIGH to LOW

Transient

HIGH/LOW to HIGH

Bus stable

Bus to high impedance

Bus change

High impedance to stable bus
APB bus signals

- **PCLK**
  - Clock
- **PADDR**
  - Address on bus
- **PWRITE**
  - 1=Write, 0=Read
- **PWDATA**
  - Data written to the I/O device. Supplied by the bus master/processor.
APB bus signals

- **PSEL**
  - Asserted if the current bus transaction is targeted to *this* device

- **PENABLE**
  - High during entire transaction *other than* the first cycle.

- **PREADY**
  - Driven by target.
  - Similar to our #ACK.
  - Indicates if the target is *ready* to do transaction.
  - Each target has its own PREADY
So what’s happening here?
Example setup

- For the next couple of slides, we will assume we have one bus master “CPU” and two slave devices (D1 and D2)
  - D1 is mapped to address 0x00001000-0x0000100F
  - D2 is mapped to addresses 0x00001010-0x0000101F
Say the CPU does a store to location 0x00001004 with no stalls
What if we want to have the LSB of this register control an LED?

We are assuming APB only gets lowest 8 bits of address here...
Reg A should be written at address 0x00001000
Reg B should be written at address 0x00001004

We are assuming APB only gets lowest 8 bits of address here...
The key thing here is that each slave device has its own read data (PRDATA) bus!
Let’s say we want a device that provides data from a switch on a read to any address it is assigned. (so returns a 0 or 1)

- PRDATA[31:0]
- PWRITE
- PENABLE
- PSEL
- PADDR[7:0]
- PCLK
- PREADY

![Mr. Switch diagram]
Device provides data from switch A if address 0x00001000 is read from. B if address 0x00001004 is read from.

- PRDATA[31:0]
- PWRITE
- PENABLE
- PSEL
- PADDR[7:0]
- PCLK
- PREADY

Switch A

Switch B
All reads read from register, all writes write...

PWDATA[31:0]

PWRITE

PENABLE

PSEL

PADDR[7:0]

PCLK

PREADY

We are assuming APB only gets lowest 8 bits of address here...
A write transfer with wait states

Setup phase begins with this rising edge
A read transfer with wait states

Setup phase begins with this rising edge
Things left out...

- There is another signal, PSLVERR (APB Slave Error) which we can drive high if things go bad.
  - We’ll just tie that to 0.

- Notice we are assuming that our device need not stall.
  - We could stall if we needed.
    - I can’t find a limit on how long, but I suspect at some point the processor would generate an error.
Verilog!

```verilog
/// APB3 BUS INTERFACE ***
input PCLK,         // clock
input PRESERN,      // system reset
input PSEL,         // peripheral select
input PENABLE,      // distinguishes access phase
output wire PREADY, // peripheral ready signal
output wire PSLVERR, // error signal
input PWRITE,       // distinguishes read and write cycles
input [31:0] PADDR, // I/O address
input wire [31:0] PWDATA, // data from processor to I/O device (32 bits)
output reg [31:0] PRDATA, // data to processor from I/O device (32-bits)

/// I/O PORTS DECLARATION ***
output reg LEDOUT,  // port to LED
input SW,           // port to switch

assign PSLVERR = 0;
assign PREADY = 1;
```

APB state machine

• IDLE
  - Default APB state

• SETUP
  - When transfer required
  - PSELx is asserted
  - Only one cycle

• ACCESS
  - PENABLE is asserted
  - Addr, write, select, and write data remain stable
  - Stay if PREADY = L
  - Goto IDLE if PREADY = H and no more data
  - Goto SETUP is PREADY = H and more data pending

We’ll spend a bit more time on this next week...
Today

- Memory-mapped I/O and bus architecture review

- ARM’s APB bus in detail

- Start on interrupts
Interrupts

Merriam-Webster:
- “to break the uniformity or continuity of”

- Informs a program of some external events
- Breaks execution flow

Key questions:
- Where do interrupts come from?
- How do we save state for later continuation?
- How can we ignore interrupts?
- How can we prioritize interrupts?
- How can we share interrupts?
I/O Data Transfer

Two key questions to determine how data is transferred to/from a non-trivial I/O device:

1. How does the CPU know when data is available?
   a. Polling
   b. Interrupts

2. How is data transferred into and out of the device?
   a. Programmed I/O
   b. Direct Memory Access (DMA)
Interrupts

Interrupt (a.k.a. exception or trap):

• An event that causes the CPU to stop executing the current program and begin executing a special piece of code called an interrupt handler or interrupt service routine (ISR). Typically, the ISR does some work and then resumes the interrupted program.

Interrupts are really glorified procedure calls, except that they:

• can occur between any two instructions
• are transparent to the running program (usually)
• are not explicitly requested by the program (typically)
• call a procedure at an address determined by the type of interrupt, not the program
Two basic types of interrupts

1/2

- Those caused by an instruction
  - Examples:
    - TLB miss
    - Illegal/unimplemented instruction
    - div by 0
  - Names:
    - Trap, exception
Two basic types of interrupts (2/2)

• Those caused by the external world
  - External device
  - Reset button
  - Timer expires
  - Power failure
  - System error

• Names:
  - interrupt, external interrupt
How it works

- Something tells the processor core there is an interrupt
- Core transfers control to code that needs to be executed
- Said code “returns” to old program
- Much harder then it looks.
  - Why?
... is in the details

- How do you figure out *where* to branch to?

- How to you ensure that you can get back to where you started?

- Don’t we have a pipeline? What about partially executed instructions?

- What if we get an interrupt while we are processing our interrupt?

- What if we are in a “critical section?”
• If you know *what* caused the interrupt then you want to jump to the code that handles that interrupt.
  - If you number the possible interrupt cases, and an interrupt comes in, you can just branch to a location, using that number as an offset (this is a branch table)
  - If you don’t have the number, you need to *poll* all possible sources of the interrupt to see who caused it.
    • Then you branch to the right code
Get back to where you once belonged

• Need to store the return address somewhere.
  - Stack *might* be a scary place.
    • *That* would involve a load/store and might cause an interrupt (page fault)!
  - So a dedicated register seems like a good choice
    • But that might cause problems later...
Snazzy architectures

- A modern processor has *many* (often 50+) instructions in-flight at once.
  - What do we do with them?
- Drain the pipeline?
  - What if one of them causes an exception?
- Punt all that work
  - Slows us down
- What if the instruction that caused the exception was executed before some other instruction?
  - What if that other instruction caused an interrupt?
Nested interrupts

• If we get one interrupt while handling another what to do?
  - Just handle it
    • But what about that dedicated register?
    • What if I’m doing something that can’t be stopped?
  - Ignore it
    • But what if it is important?
  - Prioritize
    • Take those interrupts you care about. Ignore the rest
    • Still have dedicated register problems.
Critical section

• We probably need to ignore some interrupts but take others.
  - Probably should be sure our code can’t cause an exception.
  - Use same prioritization as before.
Our processor

- Over 100 interrupt sources
  - Power on reset, bus errors, I/O pins changing state, data in on a serial bus etc.

- Need a great deal of control
  - Ability to enable and disable interrupt sources
  - Ability to control where to branch to for each interrupt
  - Ability to set interrupt priorities
    - Who wins in case of a tie
    - Can interrupt A interrupt the ISR for interrupt B?
      - If so, A can “preempt” B.

- All that control will involve memory mapped I/O.
  - And given the number of interrupts that’s going to be a pain in the rear.
## Enabling and disabling interrupt sources

- **Interrupt Set Enable and Clear Enable**
  - 0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0xE000E100 | SETENA0  | R/W  | 0     | Enable for external interrupt #0-31
  bit[0] for interrupt #0 (exception #16)
  bit[1] for interrupt #1 (exception #17)
  ...
  bit[31] for interrupt #31 (exception #47)
  Write 1 to set bit to 1; write 0 has no effect
  Read value indicates the current status |
| 0xE000E180 | CLRENA0  | R/W  | 0     | Clear enable for external interrupt #0-31
  bit[0] for interrupt #0
  bit[1] for interrupt #1
  ...
  bit[31] for interrupt #31
  Write 1 to clear bit to 0; write 0 has no effect
  Read value indicates the current enable status |
How to know where to go on an interrupt.

```assembly
23 g_pfnVectors:
24 .word _estack
25 .word Reset_Handler
26 .word NMI_Handler
27 .word HardFault_Handler
28 .word MemManage_Handler
29 .word BusFault_Handler
30 .word UsageFault_Handler
31 .word 0
32 .word 0
```

```assembly
192/ *=======================================================================
193  * Reset_Handler
194  */
195  .global Reset_Handler
196  .type  Reset_Handler, @function
197  Reset_Handler:
198  _start:
```