EECS 373
Design of Microprocessor-Based Systems

Robert Dick
University of Michigan

Lecture 6: Memory-mapped I/O review, APB, start interrupts.

24 January 2017

Slides inherited from Mark Brehob.

Outline

- Lecture flow
- Context and review
- Tuesday lab
- Assembler directives
- Hardware vs. software programming
- APB
- Caller/callee saved registers review
- Volatile keyword
- Pointers and function pointers
- Weak references
- Interrupts

Lecture flow

- Feedback: Not always clear why we are learning particular material, and jumping from topic to topic can make this worse.
- Resolutions:
  - Explain reason for each topic at transitions.
  - Context and review at start of each lecture.
- Keep on giving feedback.
  - Teaching the way I would learn best doesn't work.
    - I learn is a weird way.
  - I really do act on the feedback.

Outline

- ~~Lecture flow~~
- Context and review
- Tuesday lab
- Assembler directives
- Hardware vs. software programming
- APB
- Caller/callee saved registers review
- Volatile keyword
- Pointers and function pointers
- Weak references
- Interrupts

Context and review

- Just finished memory-mapped IO.
  - Write and read memory locations to trigger actions by peripherals.
- Approaches to design and debugging
  - Graph model
  - Get a simple version working
- Using stack in parameter passing.
- Project problem selection.
  - Were the small group meetings helpful?
  - Will require two short project proposals soon.
- Started on APB
  - Did a simple example.
  - Will do a more complex example today.

Outline

- ~~Lecture flow~~
- ~~Context and review~~
- Tuesday lab
- Assembler directives
- Hardware vs. software programming
- APB
- Caller/callee saved registers review
- Volatile keyword
- Pointers and function pointers
- Weak references
- Interrupts

Tuesday lab

- Before lecture.
- I have been watching this and covering essential material previous Thursday.
- However, I will often have reinforcing or more detailed examples on Tuesday.
- When labs are all done, will compare lab medians for Tuesday lab and rest of class.
- If there is a significant difference, will adjust Tuesday lab grades.
- Don't expect a significant difference.
  - Lab staff know their stuff.
  - Do generally cover the essentials first.

Outline

- ~~Lecture flow~~
- ~~Context and review~~
- ~~Tuesday lab~~
- Assembler directives
- Hardware vs. software programming
- APB
- Caller/callee saved registers review
- Volatile keyword
- Pointers and function pointers
- Weak references
- Interrupts

Assembler directives

- Reason for covering: Some people were confused about this in lab.
- Assembler directions don't necessary generate any instructions.
- Convenience to allow more modular and organized code, e.g., .equ .
  - Generates no code.
  - Acts like a proceprocessor macro (#define) in C.
- Provide information about data to include, e.g., .word .
- Tell assembler which symbols are global, e.g., .global .
- Indicate where in memory things (code and data) should sit, e.g., .text

Assembler directives example

```
@ “#define”-like
.equ    STACK_TOP, 0x20000800
.equ    SYSREG_SOFT_RST_CR, 0xE0042030

@ Make _start externally visible (to ld).
.global  _start

@ “a”: allocatable
@ %progbits: section contains data
@ .int_vector: section name. link.ld uses this.
.section .int_vector, "a", %progbits

_start:
        .word   STACK_TOP, main
```

Outline

- ~~Lecture flow~~
- ~~Context and review~~
- ~~Tuesday lab~~
- ~~Assembler directives~~
- Hardware vs. software programming
- APB
- Caller/callee saved registers review
- Volatile keyword
- Pointers and function pointers
- Weak references
- Interrupts

Hardware vs. software programming

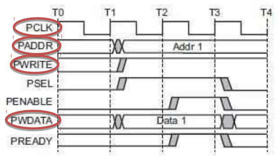
- Reasons covering
  - Common sticking point
  - A few students have had trouble with this in lab
- HDL → FPGA
  - Control which functions (gates) are implemented.
  - Control how they are connected.
- Assembly/C → ARM Cortex M-3
  - Control instruction sequences.
  - Control data to load into memory before execution.

Outline

- ~~Lecture flow~~
- ~~Context and review~~
- ~~Tuesday lab~~
- ~~Assembler directives~~
- ~~Hardware vs. software programming~~
- APB
- Caller/callee saved registers review
- Volatile keyword
- Pointers and function pointers
- Weak references
- Interrupts

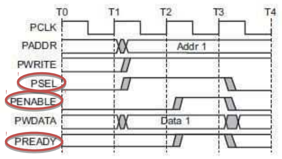
APB bus signals

- 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.

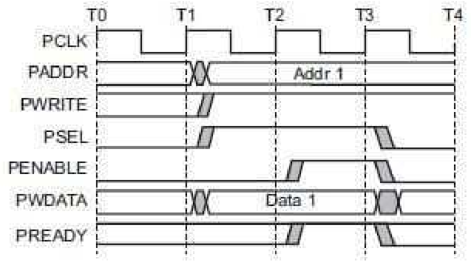


APB bus signals

- PSEL
  - Asserted if the current transaction is targeted *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 it's own PREADY



So what's happening here?

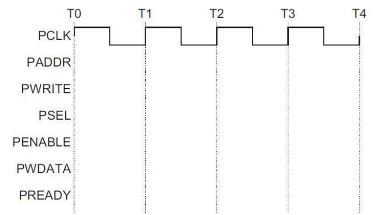


Example setup

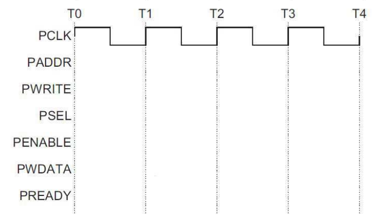
- For the next couple of slides, we will assume we have one bus master "CPU" and two slave devices (D1 and D2)
- D1 is mapped to address
  - 0x00001000-0x0000100F
  - D2 is mapped to addresses
  - 0x00001010-0x0000101F

Say the CPU does a store to location 0x00001004 with no stalls

D1

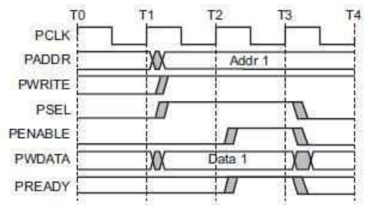


D2

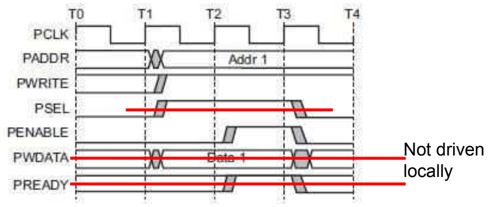


Say the CPU does a store to location 0x00001004 with no stalls

D1



D2



What if we want to have the LSB of this register control an LED?

PWDATA[31:0]

PWRITE

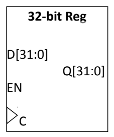
PENABLE

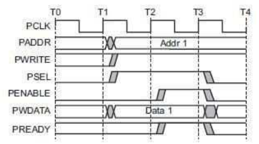
PSEL

PADDR[7:0]

PCLK

PREADY





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

PWDATA[31:0]

PWRITE

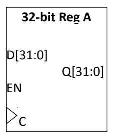
PENABLE

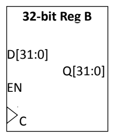
PSEL

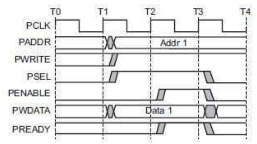
PADDR[7:0]

PCLK

PREADY

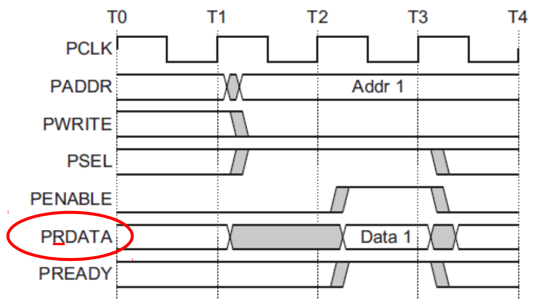






We are assuming APB only gets lowest 8 bits of address here…

Reads…



Each slave device has its own local 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

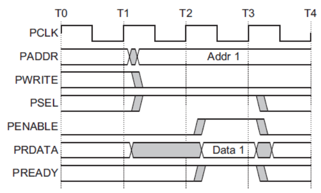
PENABLE

PSEL

PADDR[7:0]

PCLK

PREADY



Device provides data from switch A if address 0x00001000 is read from. B if address 0x00001004 is read from

PRDATA[31:0]

PWRITE

PENABLE

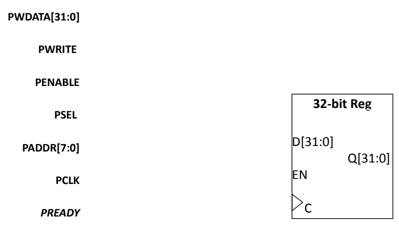
PSEL

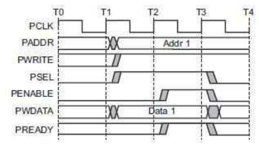
PADDR[7:0]

PCLK

PREADY

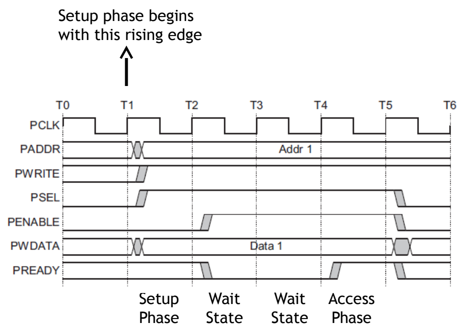
All reads read from register, all writes write…



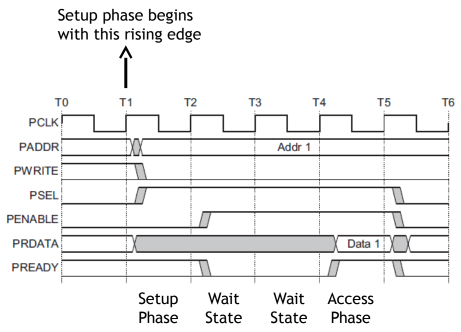


We are assuming APB only gets lowest 8 bits of address here…

A write transfer with wait states



A read transfer with wait states



Errors and stalling

- There is another signal, PSLVERR (APB Slave Error) which we can drive high if things go bad.
  - Nothing will go wrong with our device: ground it.
- Notice we are assuming that our device need not stall.
  - Could stall if we needed.
  - If you need more than a few extra cycles, generally means your design should change.

Verilog

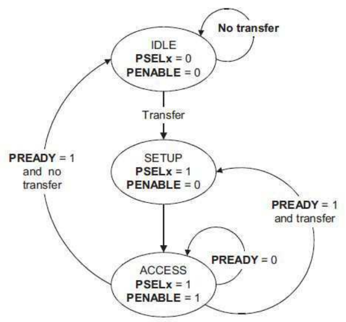
```
/*** APB3 BUS INTERFACE ***/
input PCLK,                    // clock
input PRESERN,                 // system reset
input PSEL,                    // peripheral select
input PENABLE,                 // distinguishes access phase
output wire PREADY,            // peripheral ready signal
output wire 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



Outline

- ~~Lecture flow~~
- ~~Context and review~~
- ~~Tuesday lab~~
- ~~Assembler directives~~
- ~~Hardware vs. software programming~~
- ~~APB~~
- Caller/callee saved registers review
- Volatile keyword
- Pointers and function pointers
- Weak references
- Interrupts

Caller/callee saved registers review

Reason for covering: Some people didn't understand this and it is important for the project.

Some code

function_call()
- This can walk all over r0-r3.
- If this ends up needing a veneer, the linker might insert code clobbering r12.
- Preserve r0-r3 and r12 if we'll read before write after the call.

More code

Caller/callee saved registers review

Reason for covering: Some people didn't understand this and it is important for the project.

Was just called
- I'm allowed to clobber r0-r3.
- Safe to clobber r12, too, because linker may have already clobbered it.
- Not worrying about special registers >r12.
- Need to save/restore everything else if it will be written: r5-r11

Outline

- ~~Lecture flow~~
- ~~Context and review~~
- ~~Tuesday lab~~
- ~~Assembler directives~~
- ~~Hardware vs. software programming~~
- ~~APB~~
- ~~Caller/callee saved registers review~~
- Volatile keyword
- Pointers and function pointers
- Weak references
- Interrupts

Volatile keyword

Reason for covering: You need this to safely write C code that plays with IO devices.

Definition: this value may be changed by something outside this program.

Examples
- `#define LED_ADDR ((volatile const unsigned *)(8))`
- `volatile const unsigned *led_addr = 0x8;`

Otherwise, compiler might optimize away actual memory accesses.

What's volatile? The pointer or the value pointed to?
http://cdecl.org is great!

Outline

- ~~Lecture flow~~
- ~~Context and review~~
- ~~Tuesday lab~~
- ~~Assembler directives~~
- ~~Hardware vs. software programming~~
- ~~APB~~
- ~~Caller/callee saved registers review~~
- ~~Volatile keyword~~
- Pointers and function pointers
- Weak references
- Interrupts

Pointers and function pointers

- Reason for covering: Function pointers let you esentially pass code around dynamically among functions and build vector tables in C.
- Pointers
  - Type-safe addresses.
  - Avoid void * unless really needed.
  - When would you use this?
  - The type of the object cannot be known at compile time.

Void *, a short illustrative script

Compiler: Excuse me, sir. May I suggest using a round peg?

Programmer: Shut up! I don't care! Just do it!

Compiler: As you wish, sir.

OS: Where would you like your 10GB core dump file delivered?

Function pointers

```
// Can use for generic functions.
int apple_checker(const void *x);

int orange_checker(const void *x);

int check_stuff(void *stuff_array,
     int (*checker)(const void *);

// Can use for jump tables.
void (*func_ptr[3]) = {func1, func2, func3};
```

Outline

- ~~Lecture flow~~
- ~~Context and review~~
- ~~Tuesday lab~~
- ~~Assembler directives~~
- ~~Hardware vs. software programming~~
- ~~APB~~
- ~~Caller/callee saved registers review~~
- ~~Volatile keyword~~
- ~~Pointers and function pointers~~
- Weak references
- Interrupts

Weak references

- Reason for covering: A trick to conditionally call functions that may be useful in Lab 4 and your projects.

What does a weak symbol imply?
- Provides a default entry in a function vector.
- Why useful? Allows override at link time.

What does a call through a weak symbol imply?
- If the symbol exists, call the function.
- If not, do nothing.
- Why useful? Allows link-time conditional calls without recompilation.
- Especially useful for large projects using libraries and multiple build versions.

Outline

- ~~Lecture flow~~
- ~~Context and review~~
- ~~Tuesday lab~~
- ~~Assembler directives~~
- ~~Hardware vs. software programming~~
- ~~APB~~
- ~~Caller/callee saved registers review~~
- ~~Volatile keyword~~
- ~~Pointers and function pointers~~
- ~~Weak references~~
- 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):
- Makes CPU stop executing the current program and begin executing a an **interrupt handler** or **interrupt service routine** (ISR). ISR does something and allows program to resume.

Similar to procedure calls, but
- **can occur between any two instructions**
- are transparent to the running program (usually)
- are not generally explicitly called by program
- call a procedure at an address determined by the type of interrupt, not the program

Two types of interrupts

- Those caused by an instruction
  - Examples:
    - TLB miss
    - Illegal/unimplemented instruction
    - div by 0
  - Names:
    - Trap, exception

Two basic types of interrupts

- 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?

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?"

Where

- 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
    - Ugly.

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* instructions in-flight at once.
  - What do we do with them?
- Drain the pipeline?
  - What if one of them causes an exception?
- Squash them all and restart later
  - Slows
- 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

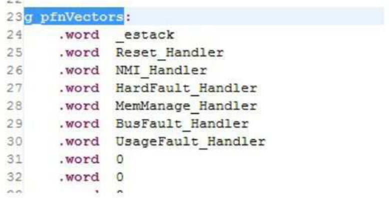
- Over 100 interrupt sources
  - Power on reset, bus errors, I/O pins changing state, data in on a serial bus etc.
- Need a great deal of control
  - Ability to enable and disable interrupt sources
  - Ability to control where to branch to for each interrupt
  - Ability to set interrupt priorities
    - 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.

Enabling and disabling interrupt sources

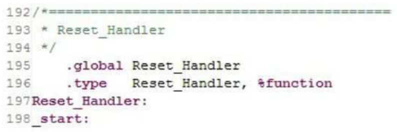
- Interrupt Set Enable and Clear Enable
  - 0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C

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.



```
g_pfnVectors:
    .word   _estack
    .word   Reset_Handler
    .word   NMI_Handler
    .word   HardFault_Handler
    .word   MemManage_Handler
    .word   BusFault_Handler
    .word   UsageFault_Handler
    .word   0
    .word   0
```



```
/*==========================================
 * Reset_Handler
 */
    .global Reset_Handler
    .type   Reset_Handler, %function
Reset_Handler:
_start:
```