Announcements

- HW2 assigned; Due on 9/23
- Questions
  - Synchronizers
  - Why separate read/write busses on APB?

Today

- Announcements
- ARM AHB-Lite
- Start on interrupts

Advanced Microcontroller Bus Architecture (AMBA)
- Advanced High-performance Bus (AHB)
- Advanced Peripheral Bus (APB)

**AHB**
- High performance
- Pipelined operation
- Burst transfers
- Multiple bus masters
- Split transactions

**APB**
- Low power
- Latched address/control
- Simple interface
- Suitable of many peripherals

Actel SmartFusion system/bus architecture
AHB-Lite supports single bus master and provides high-bandwidth operation

- Burst transfers
- Single clock-edge operation
- Non-tri-state implementation
- Configurable bus width

AHB-Lite bus master/slave interface

- Global signals
  - HCLK
  - HRESETn
- Master out/slave in
  - HADDR (address)
  - HWDATA (write data)
  - Control
    - HWRITE
    - HSIZE
    - HBURST
    - HPROT
    - HTRANS
    - HMASTLOCK
- Slave out/master in
  - HRDATA (read data)
  - HREADY
  - HRESP

AHB-Lite signal definitions

- Global signals
  - HCLK: the bus clock source (rising-edge triggered)
  - HRESETn: the bus (and system) reset signal (active low)
- Master out/slave in
  - HADDR[31:0]: the 32-bit system address bus
  - HWDATA[31:0]: the system write data bus
  - Control
    - HWRITE: indicates transfer direction (Write=1, Read=0)
    - HSIZE[2:0]: indicates size of transfer (byte, halfword, or word)
    - HBURST[2:0]: indicates single or burst transfer (1, 4, 8, 16 beats)
    - HPROT[3:0]: provides protection information (e.g. I or D; user or handler)
    - HTRANS: indicates current transfer type (e.g. idle, busy, nonseq, seq)
    - HMASTLOCK: indicates a locked (atomic) transfer sequence
- Slave out/master in
  - HRDATA[31:0]: the slave read data bus
  - HREADY: indicates previous transfer is complete
  - HRESP: the transfer response (OKAY=0, ERROR=1)

Key to timing diagram conventions

- Timing diagrams
  - Clock
  - Stable values
  - Transitions
  - High-impedance
- Signal conventions
  - Lower case ‘n’ denote active low (e.g. RESETn)
  - Prefix ‘H’ denotes AHB
  - Prefix ‘P’ denotes APB

Basic read and write transfers with no wait states

Read transfer with two wait states

Two wait states added by slave by asserting HREADY low
Valid data produced
Write transfer with one wait state

- One wait state added by slave by asserting HREADY low
- Valid data held stable

Wait states extend the address phase of next transfer

- Address stage of the next transfer is also extended
- One wait state added by slave by asserting HREADY low

Transfers can be of four types (HTRANS[1:0])

- **IDLE (b00)**
  - No data transfer is required
  - Slave must OKAY w/o waiting
  - Slave must ignore IDLE
- **BUSY (b01)**
  - Insert idle cycles in a burst
  - Burst will continue afterward
  - Address/control reflects next transfer in burst
  - Slave must OKAY w/o waiting
  - Slave must ignore BUSY
- **NONSEQ (b10)**
  - Indicates single transfer or first transfer of a burst
  - Address/control unrelated to prior transfers
- **SEQ (b11)**
  - Remaining transfers in a burst
  - Addr = prior addr + transfer size

A four beat burst with master busy and slave wait

- Master busy indicated by HTRANS[1:0]
- One wait state added by slave by asserting HREADY low

Controlling the size (width) of a transfer

- HSIZE[2:0] encodes the size
- The cannot exceed the data bus width (e.g. 32-bits)
- HSIZE + HBURST is determines wrapping boundary for wrapping bursts
- HSIZE must remain constant throughout a burst transfer

Controlling the burst beats (length) of a transfer

- Burst of 1, 4, 8, 16, and undef
- HBURST[2:0] encodes the type
- Incremental burst
- Wrapping bursts
  - 4 beats x 4-byte words wrapping
  - Wraps at 16-byte boundary
  - E.g. 0x04, 0x08, 0x0c, 0x10, ...
- Bursts must not cross 1KB address boundaries
A four beat wrapping burst (WRAP4)

A four beat incrementing burst (INCR4)

An eight beat wrapping burst (WRAP8)

An eight beat incrementing burst (INCR8) using half-word transfers

An undefined length incrementing burst (INCR)

Multi-master AHB-Lite requires a multi-layer interconnect

- AHB-Lite is single-master
- Multi-master operation
  - Must isolate masters
  - Each master assigned to layer
  - Interconnect arbitrates slave accesses
- Full crossbar switch often unneeded
  - Slaves 1, 2, 3 are shared
  - Slaves 4, 5 are local to Master 1
Today

• Announcements
• ARM AHB-Lite
• Start on interrupts

Interruptions

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)

Interruptions

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.

Interruptions 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

(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?”

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

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.
- What about instructions that shouldn’t be interrupted?

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>0xE000E100</th>
<th>SETENA0</th>
<th>R/W</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E11C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xE000E180</td>
<td>CLREN0A</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>0xE000E19C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enable for external interrupt 0-31
b[0] for interrupt 0 (exception #16)
b[1] for interrupt 1 (exception #17)
...
b[31] for interrupt 31 (exception #47)
Write 1 to set bit to 1; write 0 has no effect
Read value indicates the current status

Clear enable for external interrupt 0-31
b[0] for interrupt 0
b[1] for interrupt 1
...
b[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
  .word 0
  .word Reset_Handler
  .word BusFault_Handler
  .word UsageFault_Handler
  .word 0
  .word 0
```

```assembly
  # Reset Handler
  .type Reset_Handler, @function
  Reset_Handler:
  .start:
```