Lecture 5: Memory and Peripheral Busses

Adapted from Prabal Dutta (prabal@umich.edu)
Announcements

• The QuickRef guide seems to have errors!
  – move wide should be mov<w > <Rd>,#imm16
  – Same for add wide and sub wide!

• Knight Rider
  – http://youtu.be/WxE2xWZNfOc
  – http://youtu.be/HFqLOHlry7s

• From Etherpad
  – What range of memory is always available to write and read on the SmartFusion Chip?
What are the contents of typical linker script?

```c
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(main)

MEMORY
{
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

SECTIONS
{
  .text :
  {
    . = ALIGN(4);
    *(.text*)
    . = ALIGN(4);
    _etext = .;
  } >ram
}
end = .;
```
SECTIONS
{
  .text :
  {
    CREATE_OBJECT_SYMBOLS
    __text_load = LOADADDR(.text);
    __text_start = .;
    *(.isr_vector)
    *(.text .text.* .gnu.linkonce.t.*)
    *(.plt)
    *(.gnu.warning)
    *(.glue_7t) *(.glue_7) *(.fp11_veneer)
    . = ALIGN(0x4);
    /* These are for running static constructors and destructors under ELF. */
    KEEP (*crtbegin.o(.ctors))
    KEEP (*{EXCLUDE_FILE (*.crtend.o) .ctors})
    KEEP (*{SORT(.ctors.*)})
    KEEP (*{crtend.o(.ctors)})
    KEEP (*{EXCLUDE_FILE (*.crtend.o) .ctors})
    KEEP (*{SORT(.ctors.*)})
    KEEP (*{crtend.o(.ctors)})
    *(.rodata .rodata.* .gnu.linkonce.r.*)
    *(.ARM.extab* .gnu.linkonce.armextab.*)
    *(.gcc_except_table)
    *(.eh_frame_hdr)
    *(.eh_frame)
    KEEP (*{.init})
    KEEP (*{.fini})
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*{preinit_array})
    PROVIDE_HIDDEN (__preinit_array_end = .);
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP (*{SORT(.init_array.*)})
    KEEP (*{init_array})
    PROVIDE_HIDDEN (__init_array_end = .);
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*{fini_array})
    KEEP (*{SORT(.fini_array.*)})
    KEEP (*{fini_array})
    KEEP (*{SORT(.fini_array.*)})
    PROVIDE_HIDDEN (__fini_array_end = .);
  } >ram
  .data :
  {
    __data_load = LOADADDR (.data);
    __sidata = LOADADDR (.data);
    __data_start = .;
    _sidata = .;
    KEEP(*{.got.plt} *.got)
    *(.shdata)
    *(.data .data.* .gnu.linkonce.d.*)
    . = ALIGN (4);
    _edata = .;
  } >ram
  .bss :
  {
    __bss_start__ = .;
    __sbss = .;
    *(.shbss)
    *(.bss .bss.* .gnu.linkonce.b.*)
    *(COMMON)
    . = ALIGN (8);
    __bss_end__ = .;
    _end = .;
    _end = _end;
    ebss = .;
    PROVIDE(end = .);
  } >ram
  /
  /* The .stack section is only specified here in order for the linker to generate
  * an error if the ram is full.
  */
  .stack :
  {
    . = ALIGN(4);
    . += PROCESS_STACK_SIZE;
    . = ALIGN(4);
    . += MAIN_STACK_SIZE;
    . = ALIGN(4);
  } >ram
  .stab 0 (NOLOAD) :
  {
    *(.stab)
  }
}

etc...
The fun and easy Minute Quiz
Outline

- Minute quiz
- Announcements
- Memory and Memory-Mapped I/O
- Bus Architectures
- ARM APB
- ARM AHB-Lite
### System Memory Map

#### Memory Map of Cortex-M3

<table>
<thead>
<tr>
<th>Region</th>
<th>Virtual View</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Registers</td>
<td></td>
</tr>
<tr>
<td>External Memory Type 1</td>
<td></td>
</tr>
<tr>
<td>External Memory Type 0</td>
<td></td>
</tr>
<tr>
<td>eNVM Controller</td>
<td></td>
</tr>
<tr>
<td>eNVM Aux Block (spare pages)</td>
<td></td>
</tr>
<tr>
<td>eNVM Aux Block (array)</td>
<td></td>
</tr>
<tr>
<td>eNVM Spare Pages</td>
<td></td>
</tr>
<tr>
<td>eNVM Array</td>
<td></td>
</tr>
<tr>
<td>Peripherals (BB view)</td>
<td></td>
</tr>
<tr>
<td>FPGA Fabric</td>
<td></td>
</tr>
<tr>
<td>FPGA Fabric eSRAM Backdoor</td>
<td></td>
</tr>
<tr>
<td>Analog Compute Engine</td>
<td></td>
</tr>
<tr>
<td>IAP Controller</td>
<td></td>
</tr>
<tr>
<td>eFROM</td>
<td></td>
</tr>
<tr>
<td>RTC</td>
<td></td>
</tr>
<tr>
<td>MSS GPIO</td>
<td></td>
</tr>
<tr>
<td>I2C_1</td>
<td></td>
</tr>
<tr>
<td>SPI_1</td>
<td></td>
</tr>
<tr>
<td>UART_1</td>
<td></td>
</tr>
<tr>
<td>Fabric Interface Interrupt Controller</td>
<td></td>
</tr>
<tr>
<td>Watchdog</td>
<td></td>
</tr>
<tr>
<td>Timer</td>
<td></td>
</tr>
<tr>
<td>Peripheral DMA</td>
<td></td>
</tr>
<tr>
<td>Ethernet MAC</td>
<td></td>
</tr>
<tr>
<td>I2C_0</td>
<td></td>
</tr>
<tr>
<td>SPI_0</td>
<td></td>
</tr>
<tr>
<td>UART_0</td>
<td></td>
</tr>
<tr>
<td>eSRAM_0 / eSRAM_1 (BB view)</td>
<td></td>
</tr>
<tr>
<td>eSRAM_1</td>
<td></td>
</tr>
<tr>
<td>eSRAM_0</td>
<td></td>
</tr>
<tr>
<td>eNVM (Cortex-M3) Virtual View</td>
<td></td>
</tr>
<tr>
<td>eNVM (fabric) Virtual View</td>
<td></td>
</tr>
</tbody>
</table>

#### Memory Map of FPGA Fabric Master, Ethernet MAC, Peripheral DMA

- System Registers: 0xE0000000 – 0xE000FFFF
- External Memory Type 1: 0x70000000 – 0x7000FFFF
- External Memory Type 0: 0x71000000 – 0x7100FFFF
- eNVM Controller: 0xE0043000 – 0xE004FFFF
- eNVM Aux Block (spare pages): 0xE0048000 – 0xE004FFFF
- eNVM Aux Block (array): 0xE0049000 – 0xE004FFFF
- eNVM Spare Pages: 0xE004A000 – 0xE004FFFF
- eNVM Array: 0xE004B000 – 0xE004FFFF
- Peripherals (BB view): 0xE0050000 – 0xE005FFFF
- FPGA Fabric: 0xE0056000 – 0xE005FFFF
- FPGA Fabric eSRAM Backdoor: 0xE0057000 – 0xE005FFFF
- APB Extension Register: 0xE0058000 – 0xE005FFFF
- Analog Compute Engine: 0xE0059000 – 0xE005FFFF
- IAP Controller: 0xE005A000 – 0xE005FFFF
- eFROM: 0xE005B000 – 0xE005FFFF
- RTC: 0xE005C000 – 0xE005FFFF
- MSS GPIO: 0xE005D000 – 0xE005FFFF
- Fabric Interface Interrupt Controller: 0xE0060000 – 0xE006FFFF
- Watchdog: 0xE0061000 – 0xE006FFFF
- Timer: 0xE0062000 – 0xE006FFFF
- Peripheral DMA: 0xE0063000 – 0xE006FFFF
- Ethernet MAC: 0xE0064000 – 0xE006FFFF
- I2C_0: 0xE0065000 – 0xE006FFFF
- SPI_0: 0xE0066000 – 0xE006FFFF
- UART_0: 0xE0067000 – 0xE006FFFF
- eSRAM_0 / eSRAM_1 (BB view): 0xE0068000 – 0xE006FFFF
- eSRAM_1: 0xE0069000 – 0xE006FFFF
- eSRAM_0: 0xE006A000 – 0xE006FFFF
- eNVM (Cortex-M3) Virtual View: 0xE006B000 – 0xE006FFFF
- eNVM (fabric) Virtual View: 0xE006C000 – 0xE006FFFF
- Visible only to FPGA Fabric Master: 0xE006D000 – 0xE006FFFF

---

**Figure 2-4** • System Memory Map with 64 Kbytes of SRAM
Accessing memory locations from C

- Memory has an address and value
- Can equate a pointer to desired address
- Can set/get de-referenced value to change memory

```c
#define SYSREG_SOFT_RST_CR 0xE0042030

uint32_t *reg = (uint32_t *)(SYSREG_SOFT_RST_CR);

main () {
    *reg |= 0x00004000; // Reset GPIO hardware
    *reg &= ~(0x00004000);
}
```
Some useful C keywords

- **const**
  - Makes variable value or pointer parameter unmodifiable
  - `const int foo = 32;`

- **register**
  - Tells compiler to locate variables in a CPU register if possible
  - `register int x;`

- **static**
  - Preserve variable value after its scope ends
  - Does not go on the stack
  - `static int x;`

- **volatile**
  - Opposite of const
  - Can be changed in the background
  - `volatile int I;`
Outline

• Minute quiz

• Announcements

• Memory and Memory-Mapped I/O

• Bus Architectures

• ARM APB

• ARM AHB-Lite
What happens when this “instruction” executes?

```c
#include <stdio.h>
#include <inttypes.h>

#define REG_FOO 0x40000140

main () {
    uint32_t *reg = (uint32_t *)(REG_FOO);
    *reg += 3;

    printf("0x%x\n", *reg); // Prints out new value
}
```
“*reg += 3” is turned into a ld, add, str sequence

- **Load instruction**
  - A bus read operation commences
  - The CPU drives the address “reg” onto the address bus
  - The CPU indicated a read operation is in process (e.g. R/W#)
  - Some “handshaking” occurs
  - The target drives the contents of “reg” onto the data lines
  - The contents of “reg” is loaded into a CPU register (e.g. r0)

- **Add instruction**
  - An immediate add (e.g. add r0, #3) adds three to this value

- **Store instruction**
  - A bus write operation commences
  - The CPU drives the address “reg” onto the address bus
  - The CPU indicated a write operation is in process (e.g. R/W#)
  - Some “handshaking” occurs
  - The CPU drives the contents of “r0” onto the data lines
  - The target stores the data value into address “reg”
Details of the bus “handshaking” depend on the particular memory/peripherals involved.

• SoC memory/peripherals
  - AMBA AHB/APB

• NAND Flash
  - Open NAND Flash Interface (ONFI)

• DDR SDRAM
  - JEDEC JESD79, JESD79-2F, etc.
Modern embedded systems have multiple busses

Our expanded focus

Historical embedded focus
Why have so many busses?

- Many designs considerations
  - Master vs Slave
  - Internal vs External
  - Bridged vs Flat
  - Memory vs Peripheral
  - Synchronous vs Asynchronous
  - High-speed vs low-speed
  - Serial vs Parallel
  - Single master vs multi master
  - Single layer vs multi layer
  - Multiplexed A/D vs demultiplexed A/D

- Discussion: what are some of the tradeoffs?
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

Legend:
- SDD – Sigma-delta DAC
- SCB – Signal conditioning block
- PDMA – Peripheral DMA
- IAP – In-application programming
- ABPS – Active bipolar prescaler
- WDT – Watchdog Timer
- SWD – Serial Wire Debug

Microcontroller Subsystem
- ARM® Cortex™-M3
  - JTAG
  - NVIC
  - SysTick
  - MPU
  - ENVM
  - ESRAM
  - APB
  - SPI 1
  - UART 1
  - I^2^C 1

Programmable Analog
- DAC
- VersaTiles
- PLL
- Supervisor
  - PLL
  - OSC
  - RC
  - WDT
  - 32 KHz
  - RTC
  - SPI 1
  - UART 1
  - I^2^C 1
  - APB
  - EFM

FPGA Fabric
- SRAM
- APB
- EMC
- SysReg
- ENVM
- ESRAM
- 10/100 EMAC
- Timer1
- UART 2
- I^2^C 2

Sample Sequencing Engine
- ADC
- DAC (SDD)
- Comparator
- Temp. Mon.
- Volt Mon. (ABPS)
- Curr. Mon.

Post Processing Engine
- ADC
- DAC (SDD)
- Comparator
- Temp. Mon.
- Volt Mon. (ABPS)
- Curr. Mon.
Outline

• Minute quiz

• Announcements

• Memory and Memory-Mapped I/O

• Bus Architectures

• ARM AHB-Lite

• ARM APB
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

The EMC accepts single AHB transactions for reading and writing to external memory devices (EMDs). The EMC reformats single AHB transactions into the format required by the external EMD. The EMC may use multiple FCLK cycles to complete an EMD access, depending on the characteristics of the EMD and on the size of the access (word, half-word, or byte) and the width of the data bus to the EMD. An AHB access consists of an address phase and a data phase, as shown in Figure 7-2.

The EMC cannot complete EMD read and write transactions in only two FCLK cycles, so the user must configure the EMC and insert wait states in the data phase of the AHB access to complete the EMD access. Figure 7-4 shows an AHB read transaction with two wait states inserted into the data phase of the AHB transaction.

Figure 7-2 • AHB Address/Data Phase for Read Transfer

Figure 7-3 • AHB Address/Data Phase for Write Transfer

From: Actel SmartFusion Microcontroller Subsystem User’s Guide
The EMC accepts single AHB transactions for reading and writing to external memory devices (EMDs). The EMC reformats single AHB transactions into the format required by the external EMD. The EMC may use multiple FCLK cycles to complete an EMD access, depending on the characteristics of the EMD and on the size of the access (word, half-word, or byte) and the width of the data bus to the EMD. An AHB access consists of an address phase and a data phase, as shown in Figure 7-2.

The EMC cannot complete EMD read and write transactions in only two FCLK cycles, so the user must configure the EMC and insert wait states in the data phase of the AHB access to complete the EMD access.

Figure 7-4 shows an AHB read transaction with two wait states inserted into the data phase of the AHB transaction.

---

**Figure 7-4 • AHB Read Access 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
- They cannot exceed the data bus width (e.g. 32-bits)
- HSIZE + HBURST determines wrapping boundary for wrapping bursts
- HSIZE must remain constant throughout a burst transfer

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>Byte</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16</td>
<td>Halfword</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>Word</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>Doubleword</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>4-word line</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>256</td>
<td>8-word line</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1024</td>
<td></td>
</tr>
</tbody>
</table>
• 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. 0x34, 0x38, 0x3c, 0x30,…

• Bursts must not cross 1KB address boundaries

<table>
<thead>
<tr>
<th>HBURST[2:0]</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b000</td>
<td>SINGLE</td>
<td>Single burst</td>
</tr>
<tr>
<td>b001</td>
<td>INCR</td>
<td>Incrementing burst of undefined length</td>
</tr>
<tr>
<td>b010</td>
<td>WRAP4</td>
<td>4-beat wrapping burst</td>
</tr>
<tr>
<td>b011</td>
<td>INCR4</td>
<td>4-beat incrementing burst</td>
</tr>
<tr>
<td>b100</td>
<td>WRAP8</td>
<td>8-beat wrapping burst</td>
</tr>
<tr>
<td>b101</td>
<td>INCR8</td>
<td>8-beat incrementing burst</td>
</tr>
<tr>
<td>b110</td>
<td>WRAP16</td>
<td>16-beat wrapping burst</td>
</tr>
<tr>
<td>b111</td>
<td>INCR16</td>
<td>16-beat incrementing burst</td>
</tr>
</tbody>
</table>
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)
- 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
Outline

• Minute quiz
• Announcements
• Memory and Memory-Mapped I/O
• Bus Architectures
• ARM AHB-Lite
• ARM APB
APB is simpler interface than AHB-Lite and provides high-bandwidth operation

- Low-cost
- Low-power
- Low-complexity
- Low-bandwidth
- Non pipelined
- Ideal for peripherals
• PCLK: the bus clock source (rising-edge triggered)
• PRESETn: the bus (and typically system) reset signal (active low)
• PADDR: the APB address bus (can be up to 32-bits wide)
• PSELx: the select line for each slave device
• PENABLE: indicates the 2\textsuperscript{nd} and subsequent cycles of an APB xfer
• PWRITE: indicates transfer direction (Write=H, Read=L)
• PWDATA: the write data bus (can be up to 32-bits wide)
• PREADY: used to extend a transfer
• PRDATA: the read data bus (can be up to 32-bits wide)
• PSLVERR: indicates a transfer error (OKAY=L, ERROR=H)
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 if PREADY = H and more data pending
A write transfer with no wait states

Setup phase begins with this rising edge

Diagram shows:
- T0, T1, T2, T3, T4 time intervals
- PCLK, PADDR, PWRITE, PSEL, PENABLE, PWDATA, PREADY signals
- Setup Phase and Access Phase indicators
- Addr 1 and Data 1 address/data waves
Setup phase begins with this rising edge

- **T0** to **T1**: Setup Phase
- **T2** to **T3**: Wait State
- **T4** to **T5**: Wait State
- **T6**: Access Phase
A read transfer with no wait states

Setup phase begins with this rising edge

Setup Phase  Access Phase
A read transfer with wait states

Setup phase begins with this rising edge

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PADDR</td>
<td></td>
<td>Addr 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PWRITE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSEL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PENABLE</td>
<td></td>
<td></td>
<td></td>
<td>Data 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRDATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREADY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Setup Phase  Wait State  Wait State  Access Phase
Lab 3: creating a memory-mapped peripheral that lives on the AMBA APB
Questions?

Comments?

Discussion?