CPEN 311: Digital Systems Design

Introduction to System-on-Chip Design

What & why

This slide set will introduce you to System-on-Chip (SoC) Design

Most real chips are designed like this

You will see a lot more of this if you take CPEN 391

Real systems are usually *not* built from scratch.

Real systems usually contain both hardware and software.

Real systems usually revolve around an embedded processor.

Real systems are usually designed using system-level design tools.

In this slide set, we will talk about how to design real systems consisting of a processor, software, and hardware all working together

In CPEN 311, we were thinking primarily of building a system only in hardware.

- I/O pins connect to sensors, displays, etc.

But real systems, often contain hardware and software

- In SoC system, hardware is often connected to software through a memory-mapped interface or dedicated interconnect.

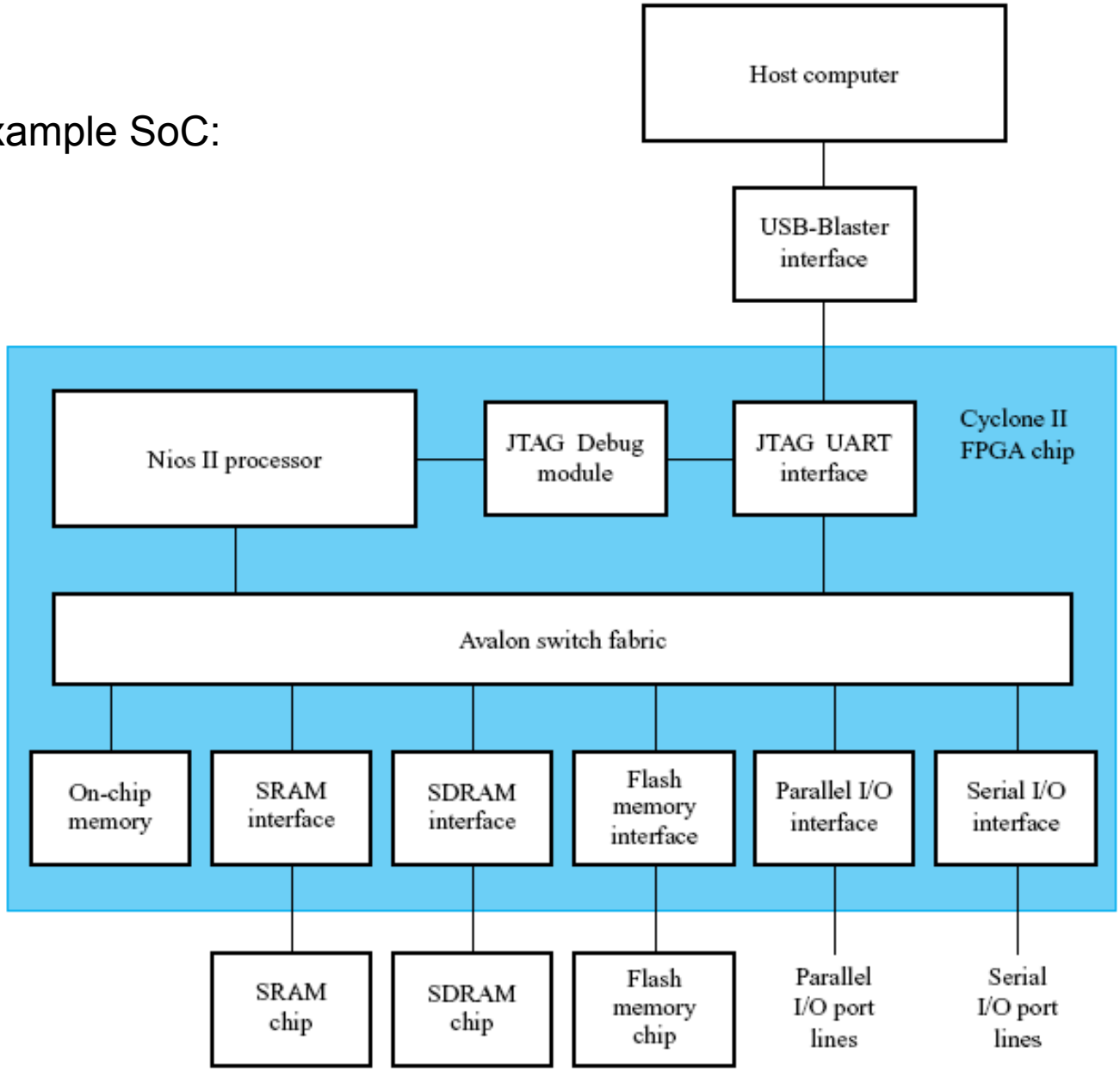
FPGA Vendors support this by providing embedded processors and tools to use them

Buzzword: System-on-Chip (SoC)

An SoC is a chip that usually consists of a processor and several embedded cores. The cores are usually connected using some sort of standard bus or other interconnect.

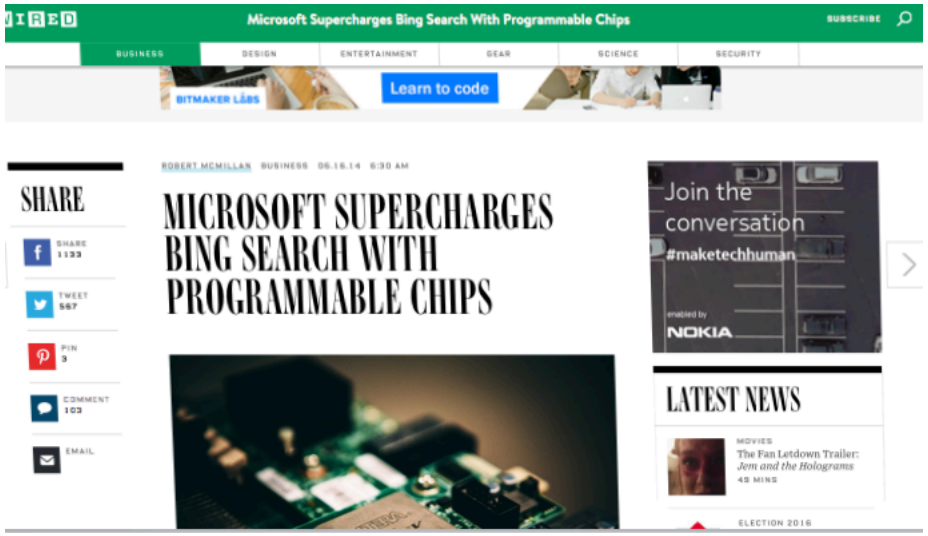
Can be implemented on an FPGA or custom chip (ASIC)

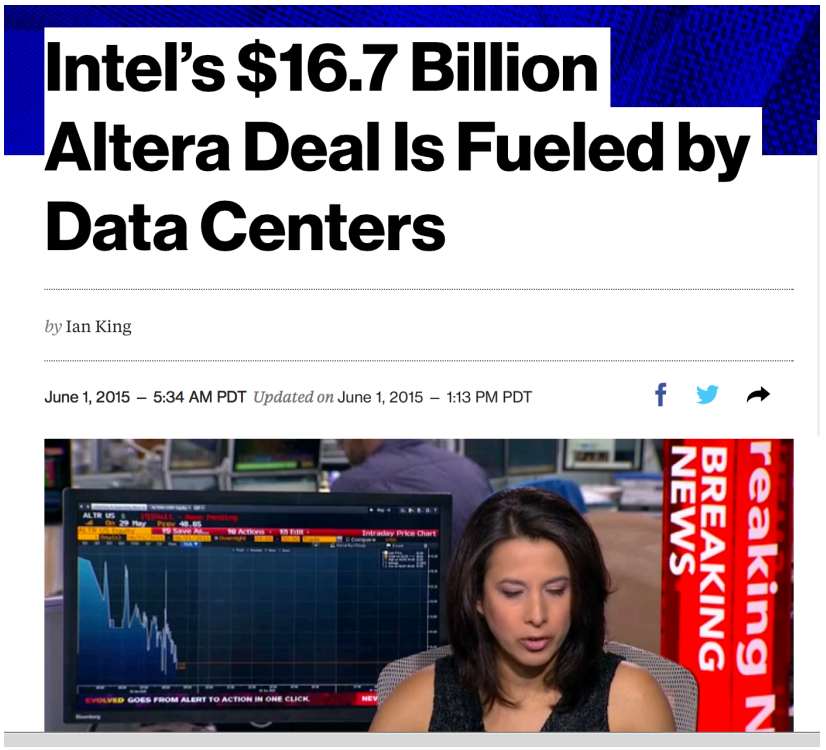
An example SoC:



Hardware Acceleration

Allows for mixed hardware/software systems:





Hot trend: FPGAs in the data-center

- Each node in a data center may have a processor and hardware fabric (based on an FPGA)
  - Programming this involves writing both software and hardware

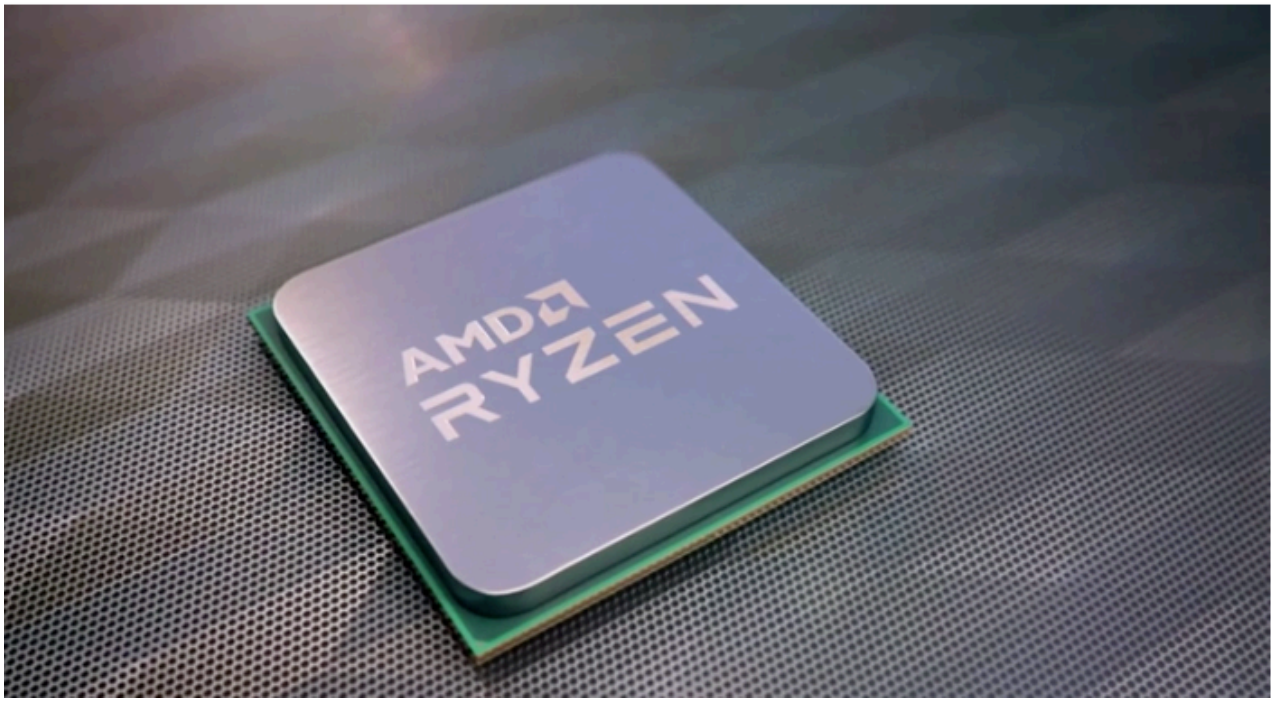
APAC OCTOBER 27, 2020 / 7:06 AM / UPDATED 10 DAYS AGO

AMD to buy chip peer Xilinx for $35 billion in data center push

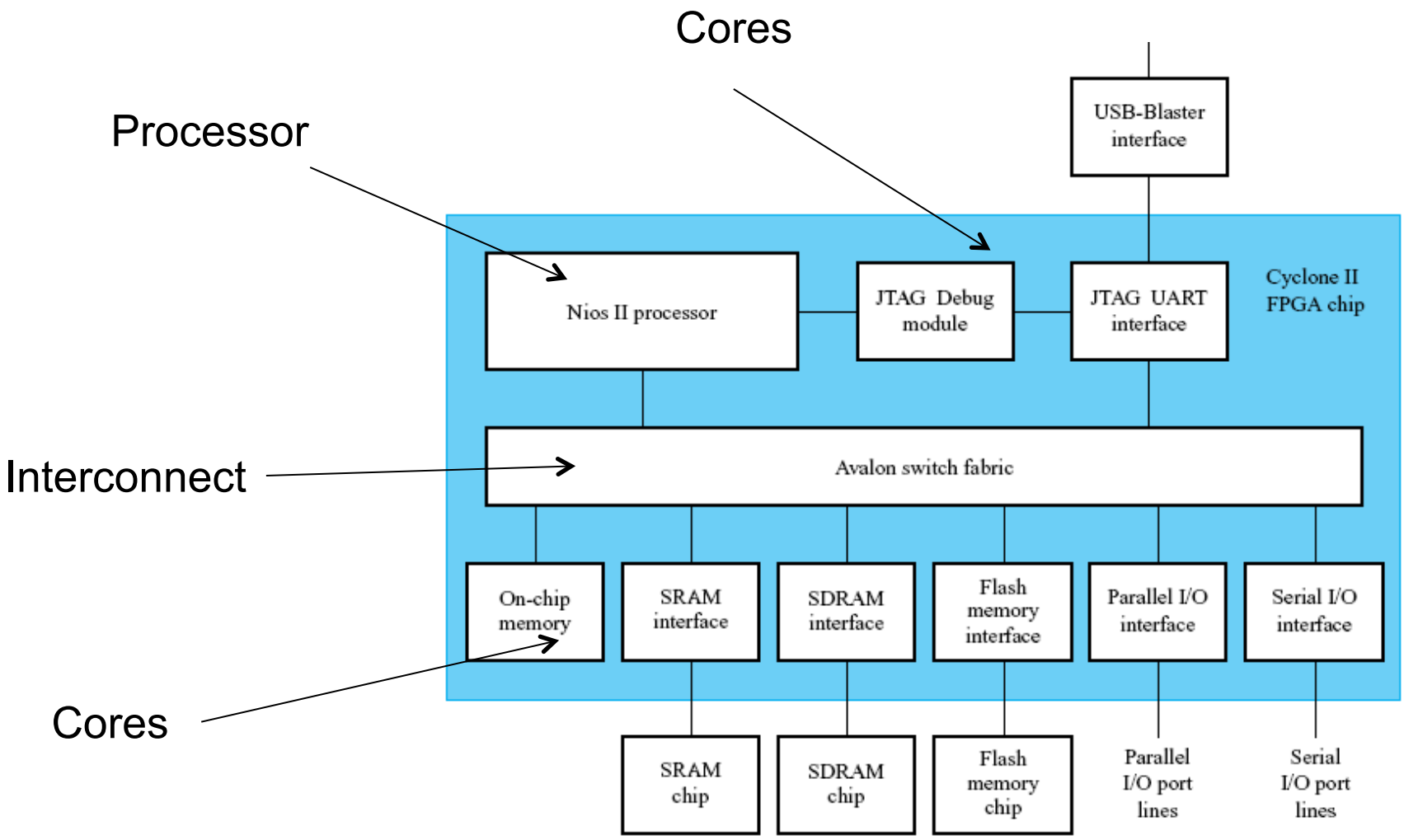
By Stephen Nellis

5 MIN READ

(Reuters) - Semiconductor designer Advanced Micro Devices Inc (AMD) said on Tuesday it would buy Xilinx Inc in a $35 billion all-stock deal, intensifying its battle with Intel Corp in the data center chip market.



Organization of an SoC



Processor Options

Recall: Implementing Systems on an FPGA

Embedded Processors can be “soft” or “hard”:

Soft: Processor implemented using the LUTs and other FPGA resources

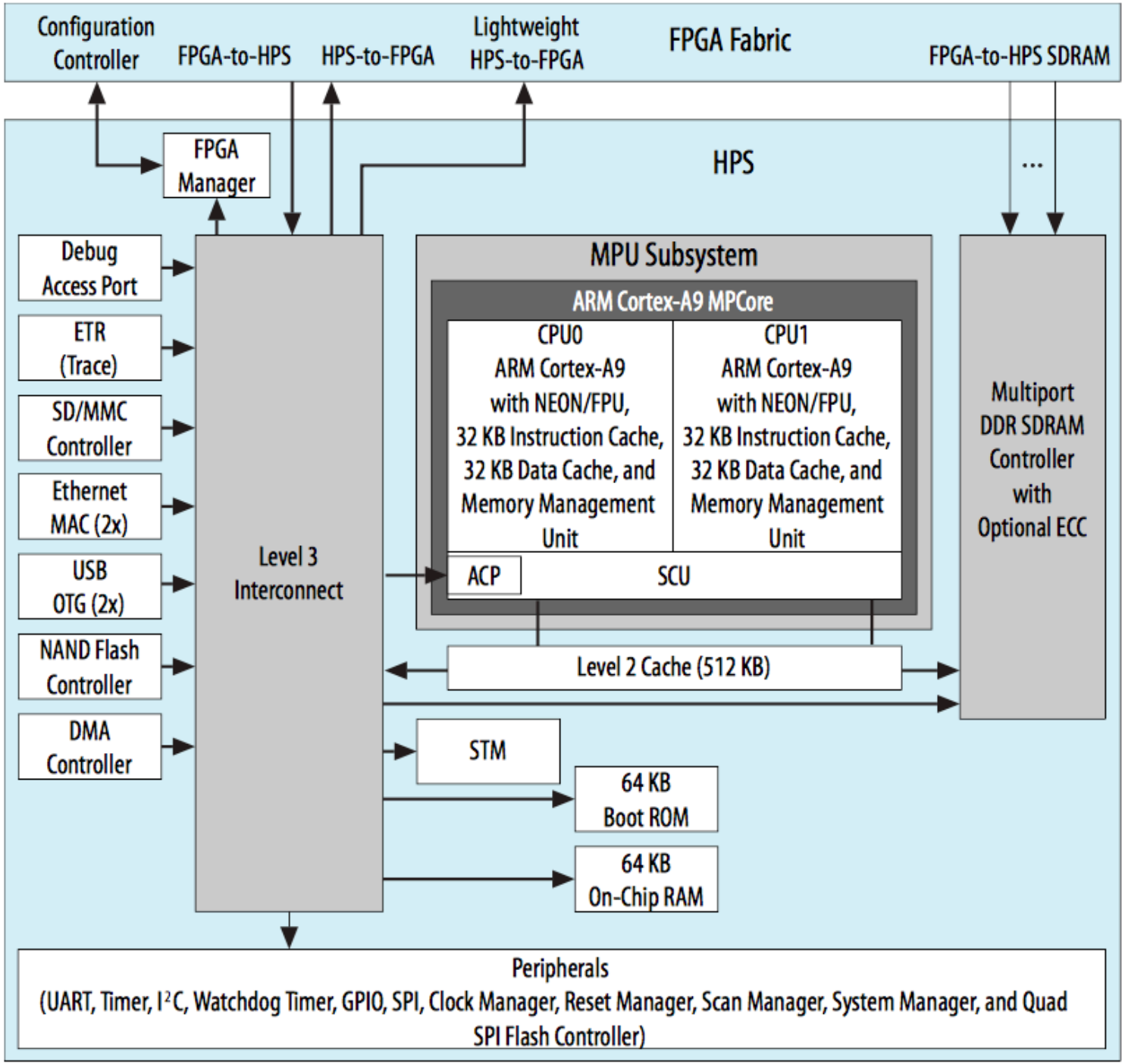
Intel: Nios-II
AMD: MicroBlaze

Hard: Hard macro embedded into an FPGA.

AMD and Intel: both have ARM Cortex processors
Latest: Quad-core 64-bit ARM Cortex A-53

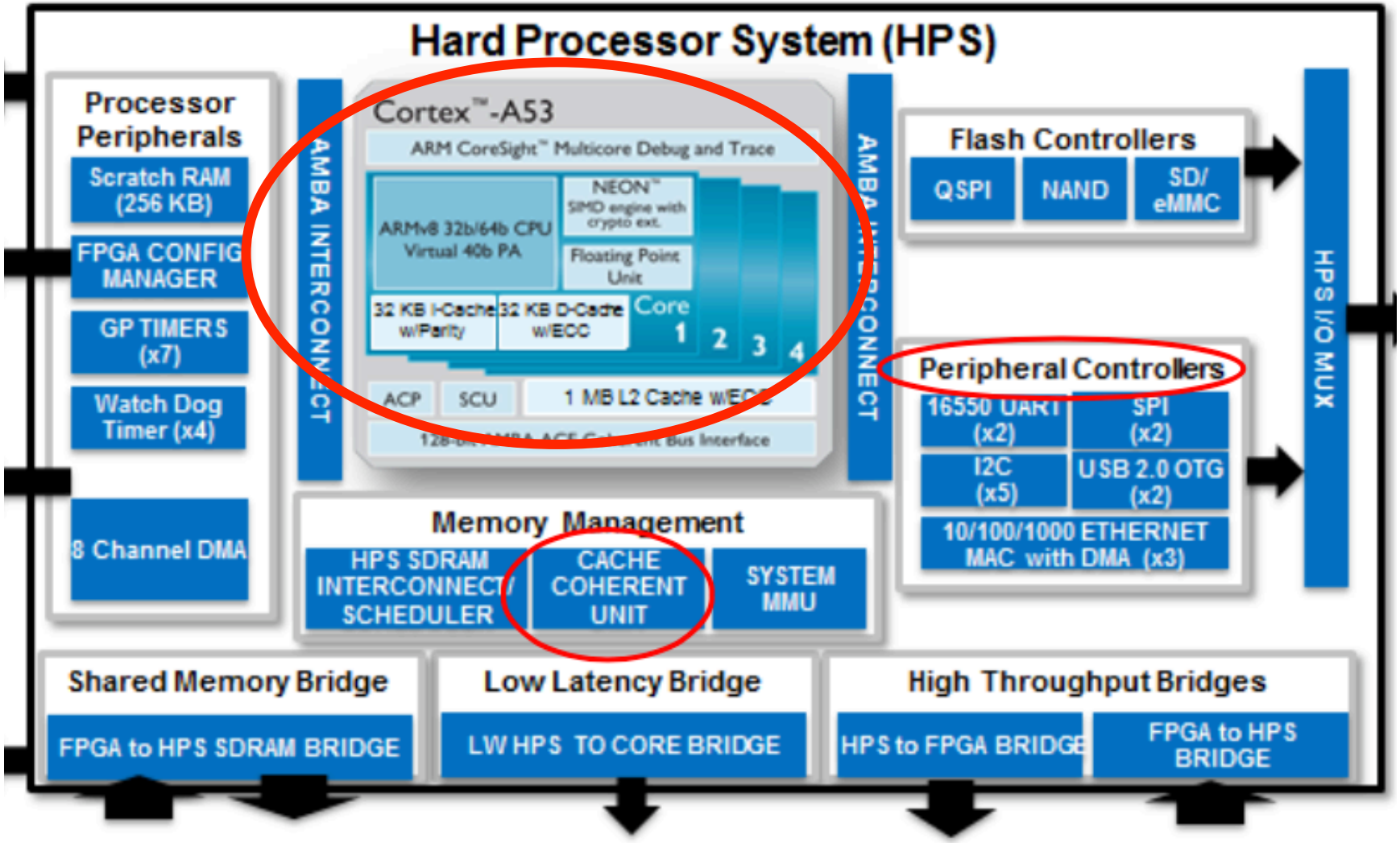
Off-chip processors can also be coupled to an FPGA

Cyclone V

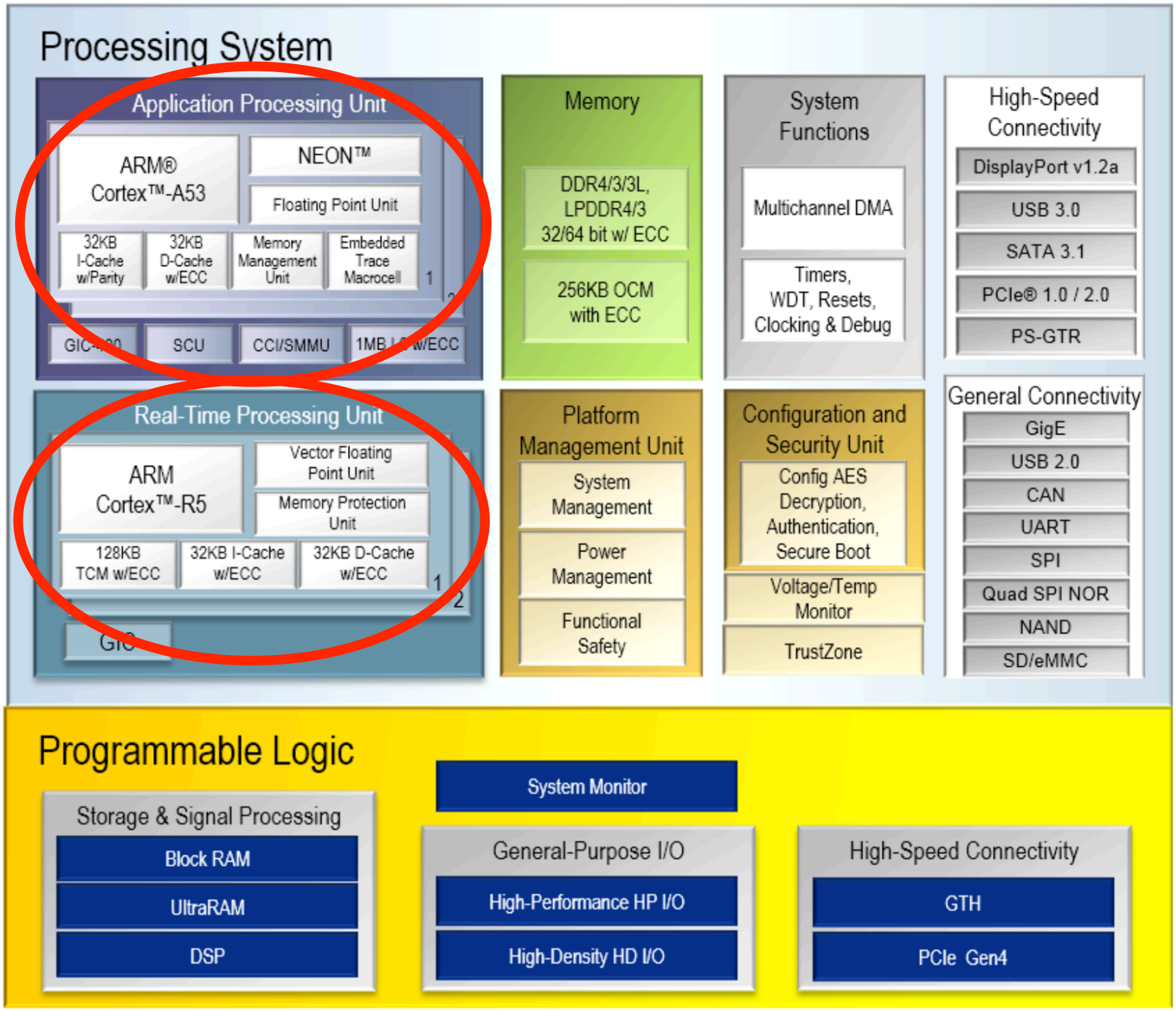


From:
Cyclone V Handbook,
Jan 2015

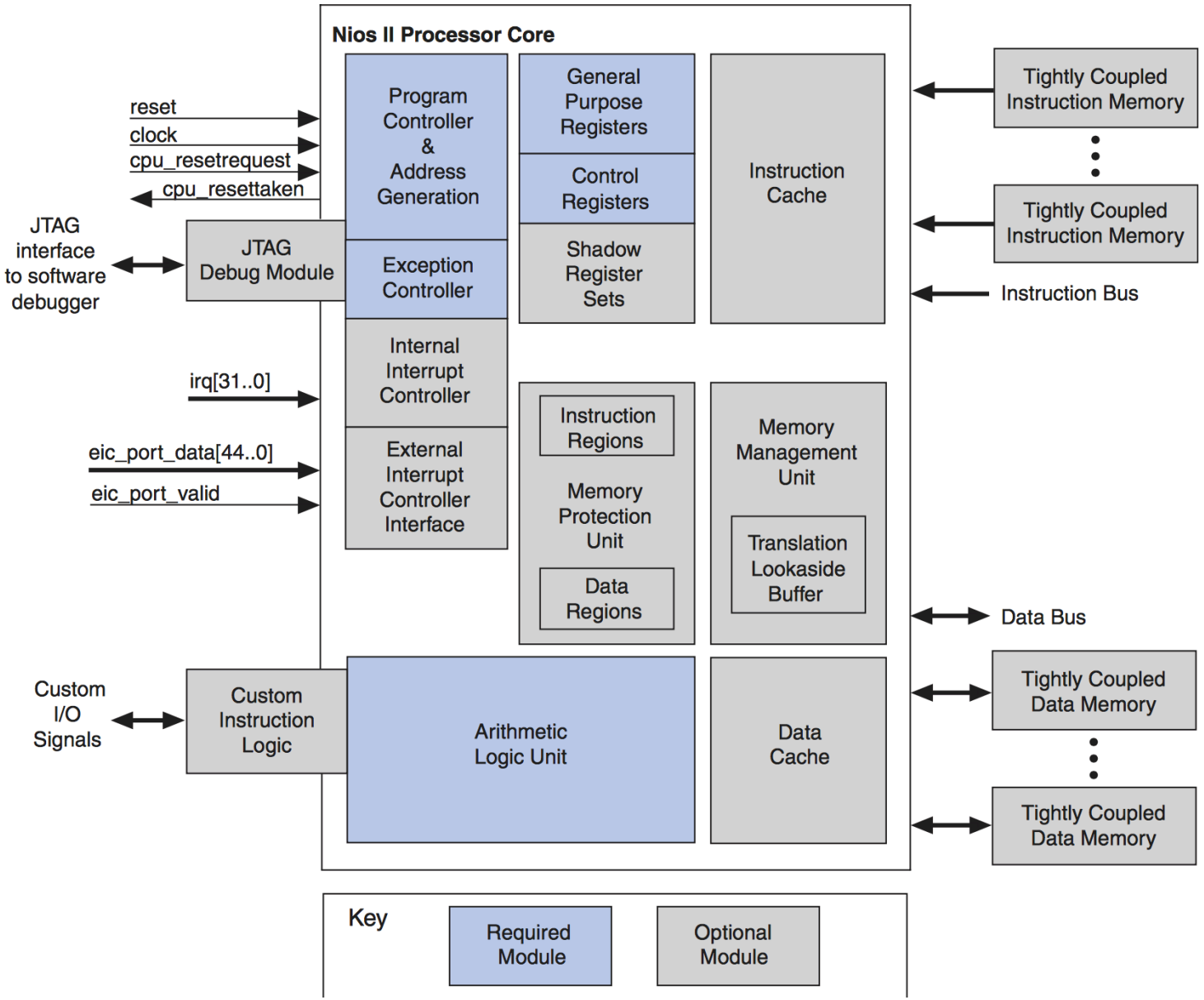
Stratix 10


Hard Processor System (HPS)
Processor Peripherals
Scratch RAM (256 KB)
FPGA CONFIG MANAGER
GP TIMERS (x7)
Watch Dog Timer (x4)
8 Channel DMA
AMBA INTERCONNECT
Cortex™-A53
ARM CoreSight™ Multicore Debug and Trace
ARMv8 32b/64b CPU Virtual 40b PA
NEON™ SIMD engine with crypto ext.
Floating Point Unit
32 KB I-Cache w/Parity
32 KB D-Cache w/ECC
Core 1 2 3 4
ACP
SCU
1 MB L2 Cache w/ECC
128-bit AMBA-ACE Coherent Bus Interface
AMBA INTERCONNECT
Flash Controllers
QSPI
NAND
SD/ eMMC
Peripheral Controllers
16550 UART (x2)
SPI (x2)
I2C (x5)
USB 2.0 OTG (x2)
10/100/1000 ETHERNET MAC with DMA (x3)
HPS I/O MUX
Memory Management
HPS SDRAM INTERCONNECT/ SCHEDULER
CACHE COHERENT UNIT
SYSTEM MMU
Shared Memory Bridge
FPGA to HPS SDRAM BRIDGE
Low Latency Bridge
LW HPS TO CORE BRIDGE
High Throughput Bridges
HPS to FPGA BRIDGE
FPGA to HPS BRIDGE

AMD Zynq UltraScale+



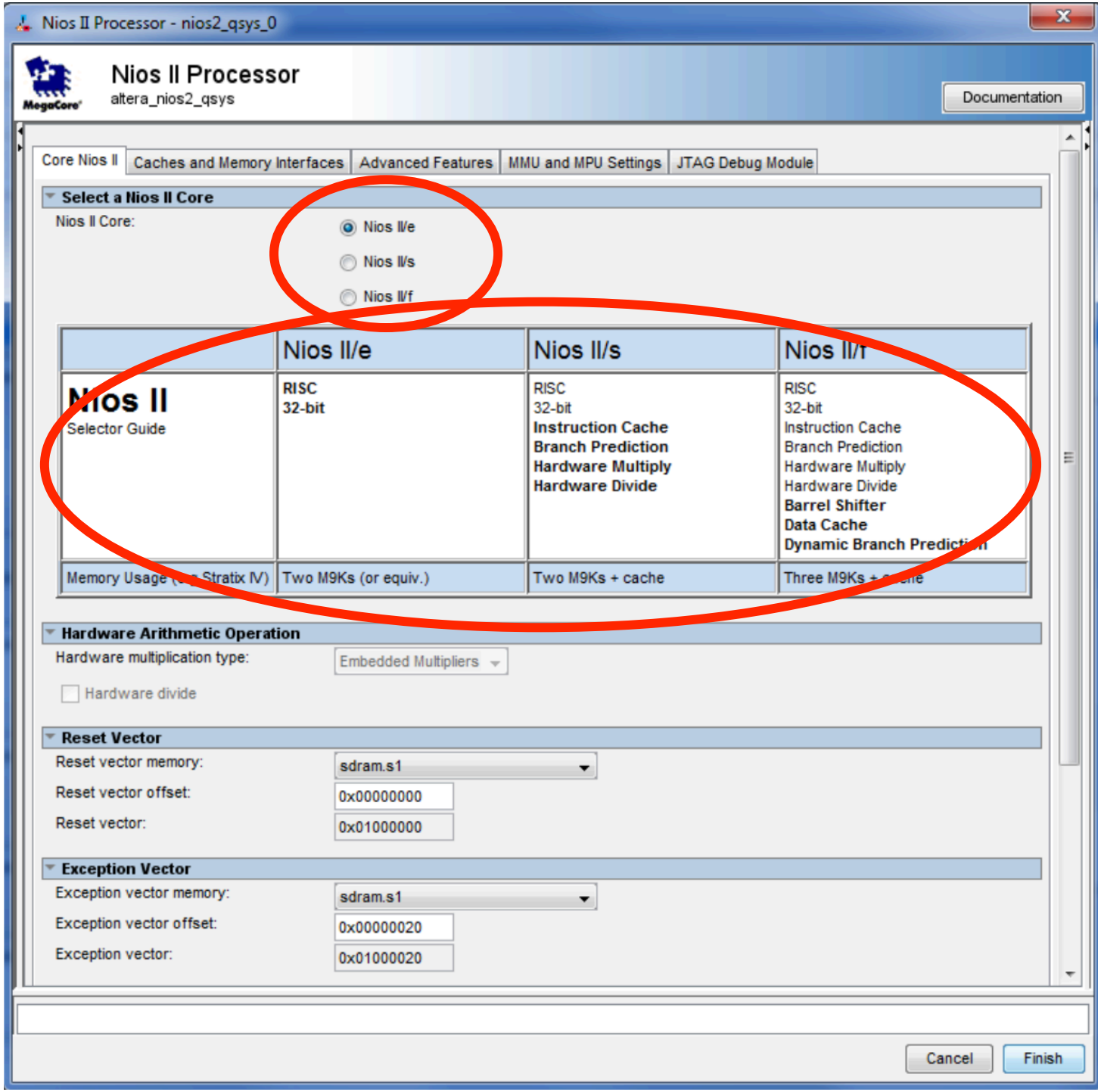
Intel NIOS II Soft-Core



From: NIOS Architecture and Programming

Several versions of NIOS II are available

Each version can be customized:



Interconnect between the
Processor and Cores

Interfacing with a CPU

- CPUs connect to devices through “interconnect”
  - Simplest is a “bus”
    - Historic, designed for off-chip devices and limited I/O pins
    - Easy to understand
  - Modern SoCs use an “interconnect fabric”
    - ARM AXI (standard), part of AMBA (AHB, APB, AXI)
    - Intel Avalon (simpler for FPGAs)
    - Wishbone
  - Others
    - Eg: IBM CoreConnect (PLB, OPB)
    - some still called a “bus” informally
      - usually not buses (esp. on FPGAs)

Masters and Servants

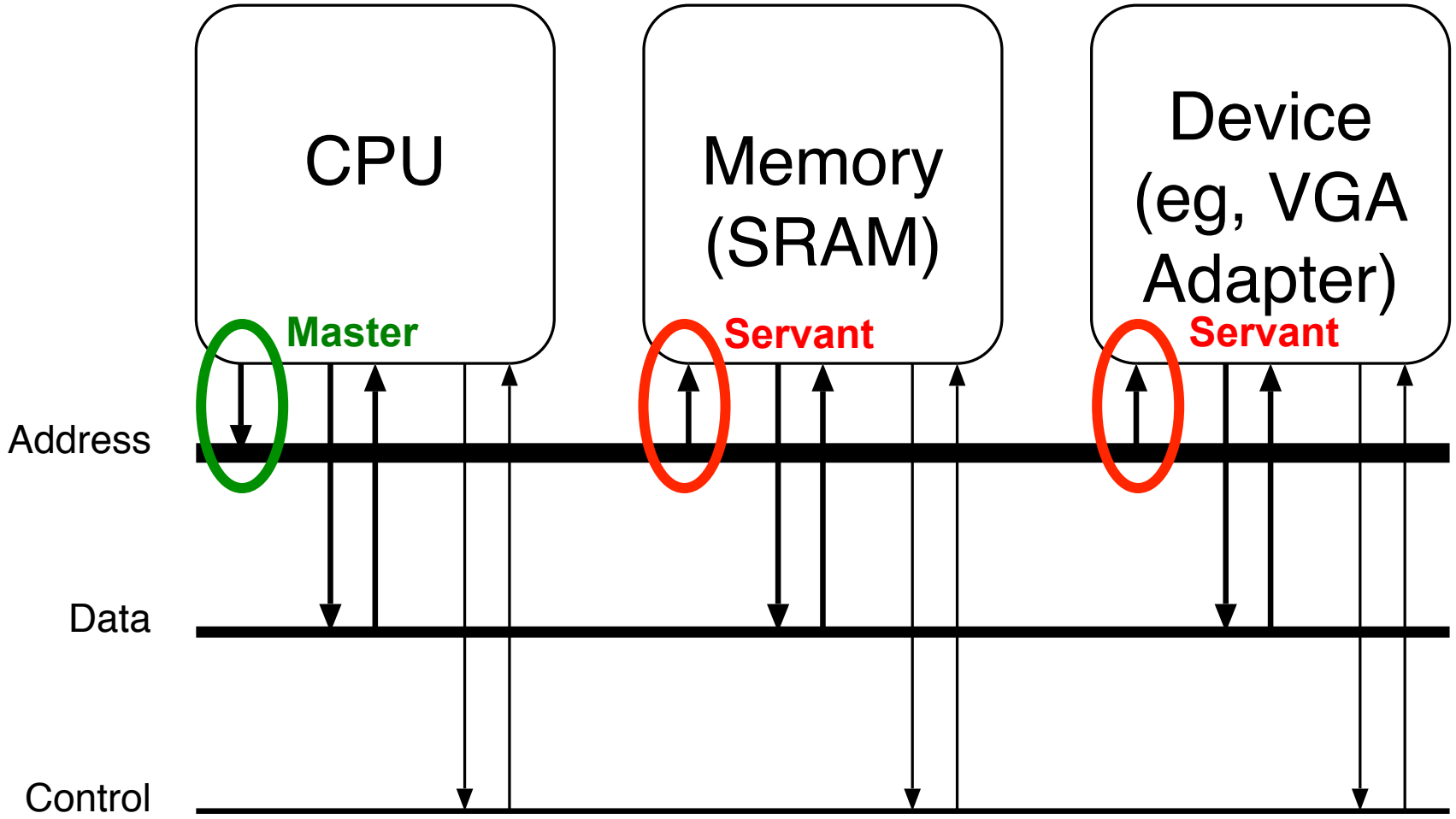
Bus protocols draw a distinction between:

- Masters
  - "requester" that initiates a transaction
  - Generates/sends an address
  - Eg: CPUs

- Servants (formerly "slave")
  - "target" that accepts a transaction
  - Receives an address, can return data as a response
  - Eg: memory

- "Slave"? Word origin: human slavery, an abhorrent practise
  - When used as a technical term, it refers back to this origin to derive meaning
  - This "re-use" desensitizes us of its true original meaning & is insensitive to those oppressed by slavery
  - Term "Servants" used here is not official/popularized; eventually "slave" will be replaced by some new term
- "Master" has other meanings/origins, such as mastery of a skill, so not offensive on its own

CPU + Bus + Memory + Peripherals



The Master-Servant abstraction

- Address (request)
  - Sent by Master, Received by 1 or more Servants
  - Typically 20-40 bits
    - 20 bits = 2^20 addresses = 1 MB
    - 40 bits = 2^40 addresses = 1 TB

- Data (request and response)
  - Off-chip, usually bidirectional wires
  - Read: Master receives, Servant sends
  - Write: Master sends, Servant receives
  - Typically 8-256 bits
    - 8 bits = small computer (eg, microwave oven, clock radio)
    - 16, 32, 64 bits = fast embedded computer
    - 128, 256 bits = fast mainstream computer

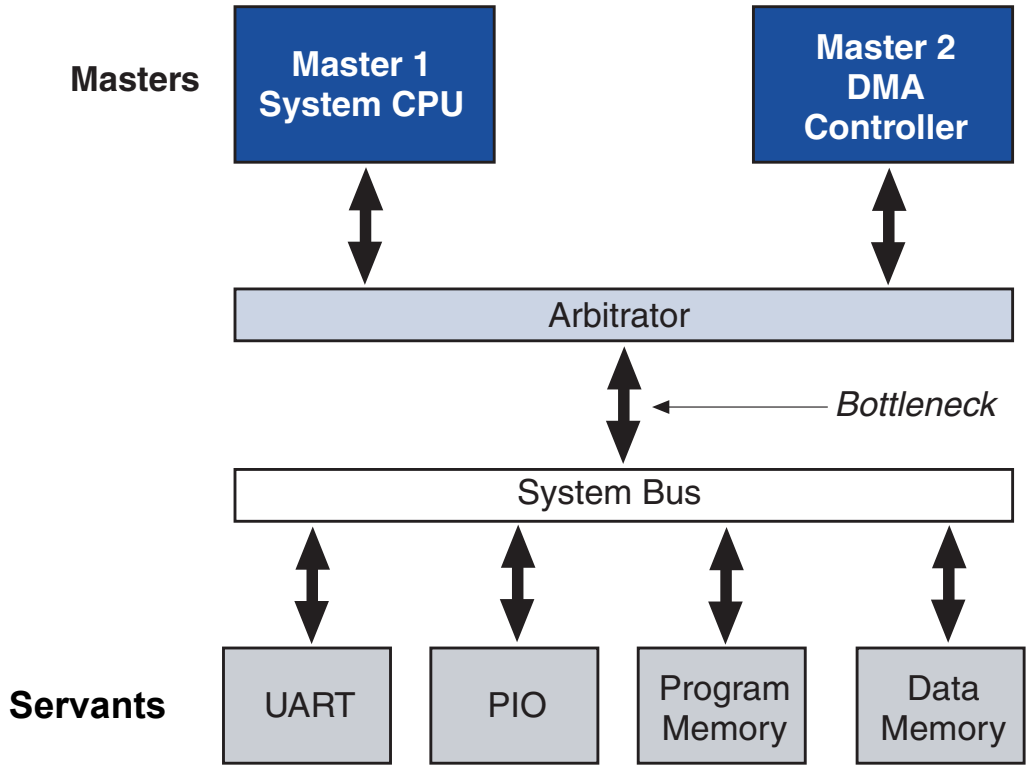
Intel's Avalon "Bus"

Rather than a single bus, connections are made through a multiplexing structure. Multiple masters can be sending data at the same time, as long as they do not access the same Servant in the same clock cycle.

Described in document: *Avalon Memory-Mapped Interface Specification*

Avalon Compared to Traditional Bus

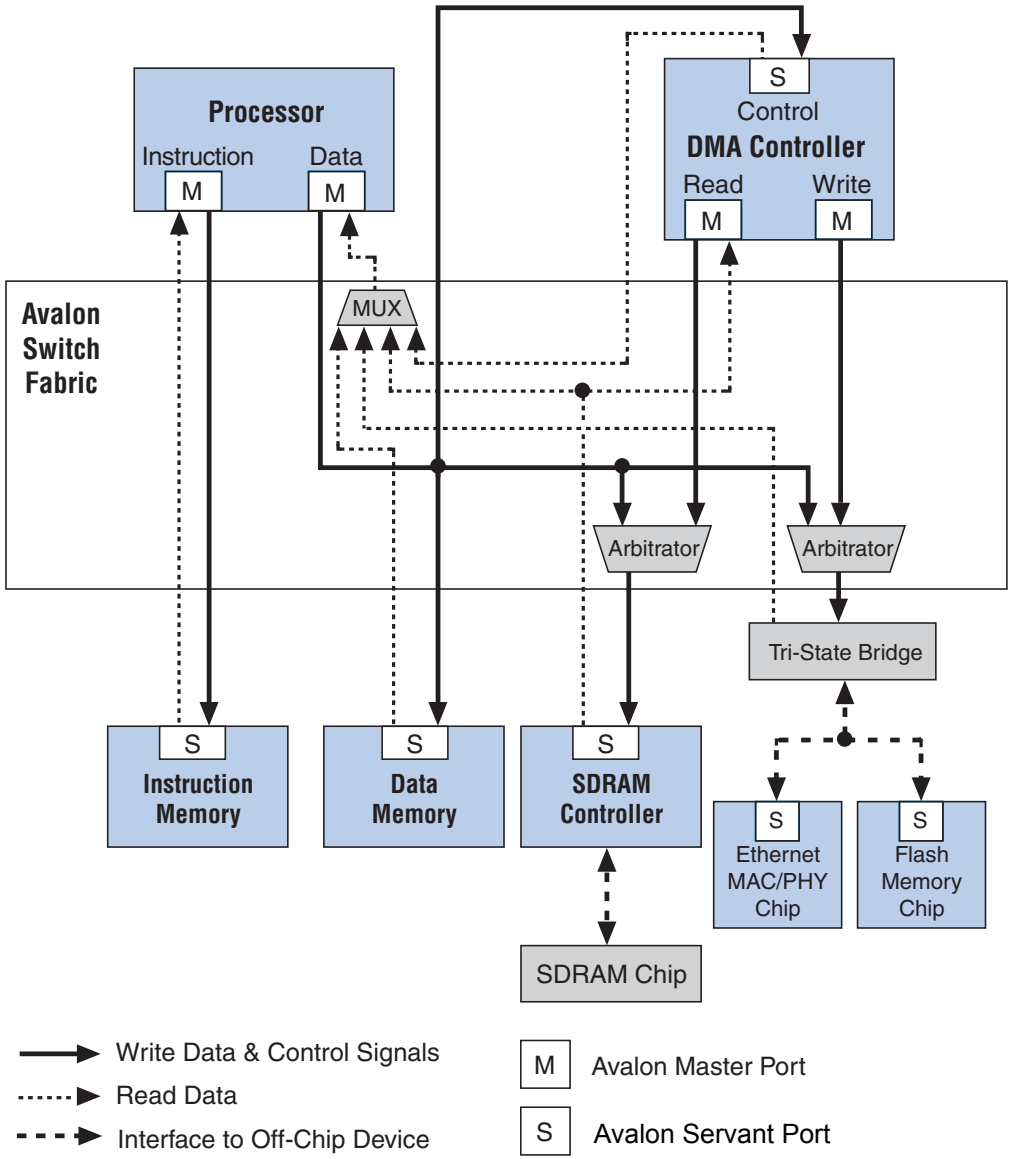
Figure 3–7. Bus Architecture in a Traditional Microprocessor System



Source: Altera

Example Avalon System

Figure 3–1. Avalon Switch Fabric Block Diagram – Example System



Don’t worry about the details here, the point is that the interconnect is mux-based.

Less prone to bottlenecks if there are multiple masters, since multiple transfers can go on at the same time.

Source: Altera

Memory Mapped Peripherals

Each peripheral attached to the bus is assigned a region of the address space.

- These address spaces are “special” in that writing or reading transfers data to or from the peripheral rather than accessing memory

- Example: Later we will show an example of a graphics accelerator:
    - Mapped to locations I/O 0x002100 to 0x002110

    - If the processor writes to an address in this region, the data will be sent to the accelerator rather than being written to memory

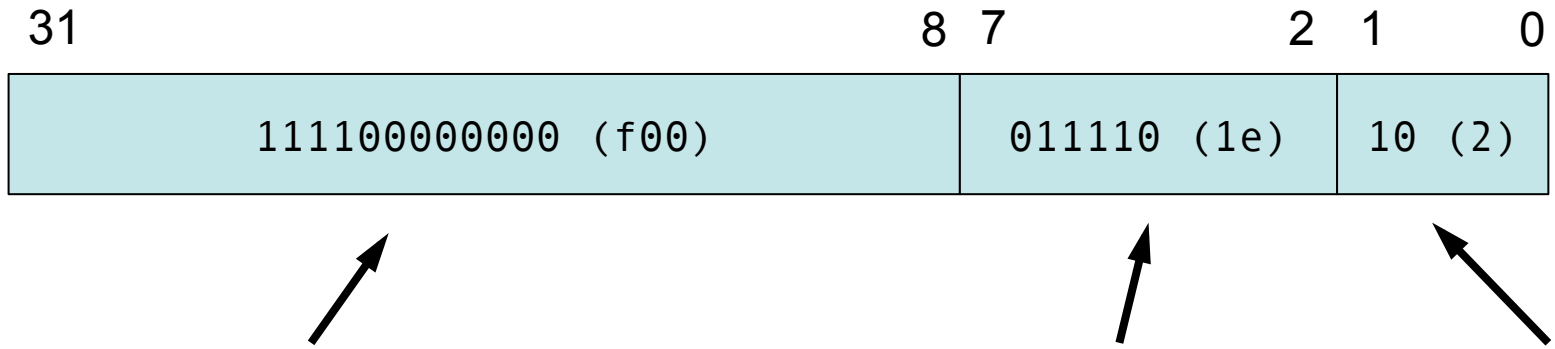
    - addressing granularity often different for CPU & interconnect!

Memory Mapping Example

Avalon 32-bit bus (e.g., in our lab), but CPU memory byte-addressable

Device memory map range: `0xf0000-0xf00ff`
(from the perspective of the CPU)

address `0xf007a = 0000·0000·0000·1111·0000·0000·0111·1010`



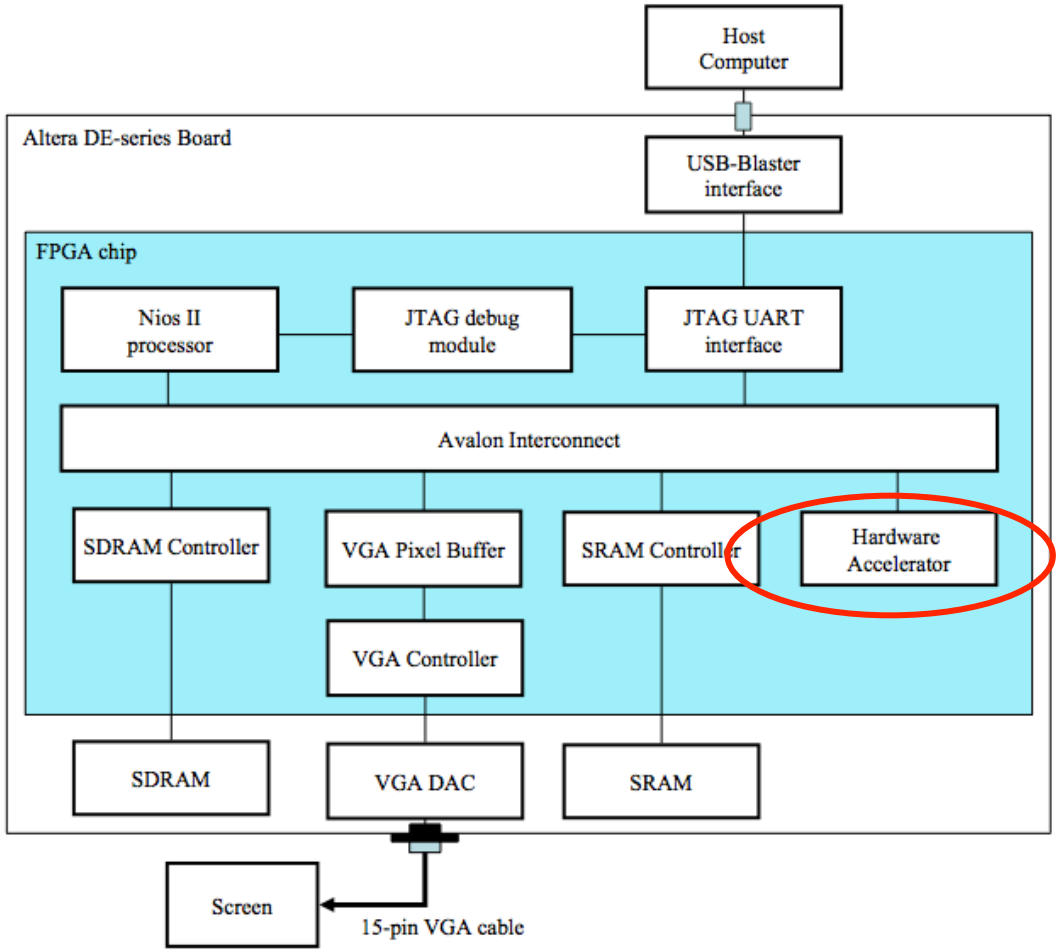
identifies the device — used by Avalon interconnect to deliver messages there

sent to the device as the address (note: 32-bit word granularity!)

sent to device as byte select (if enabled)

Ways of Interfacing a Hardware Core

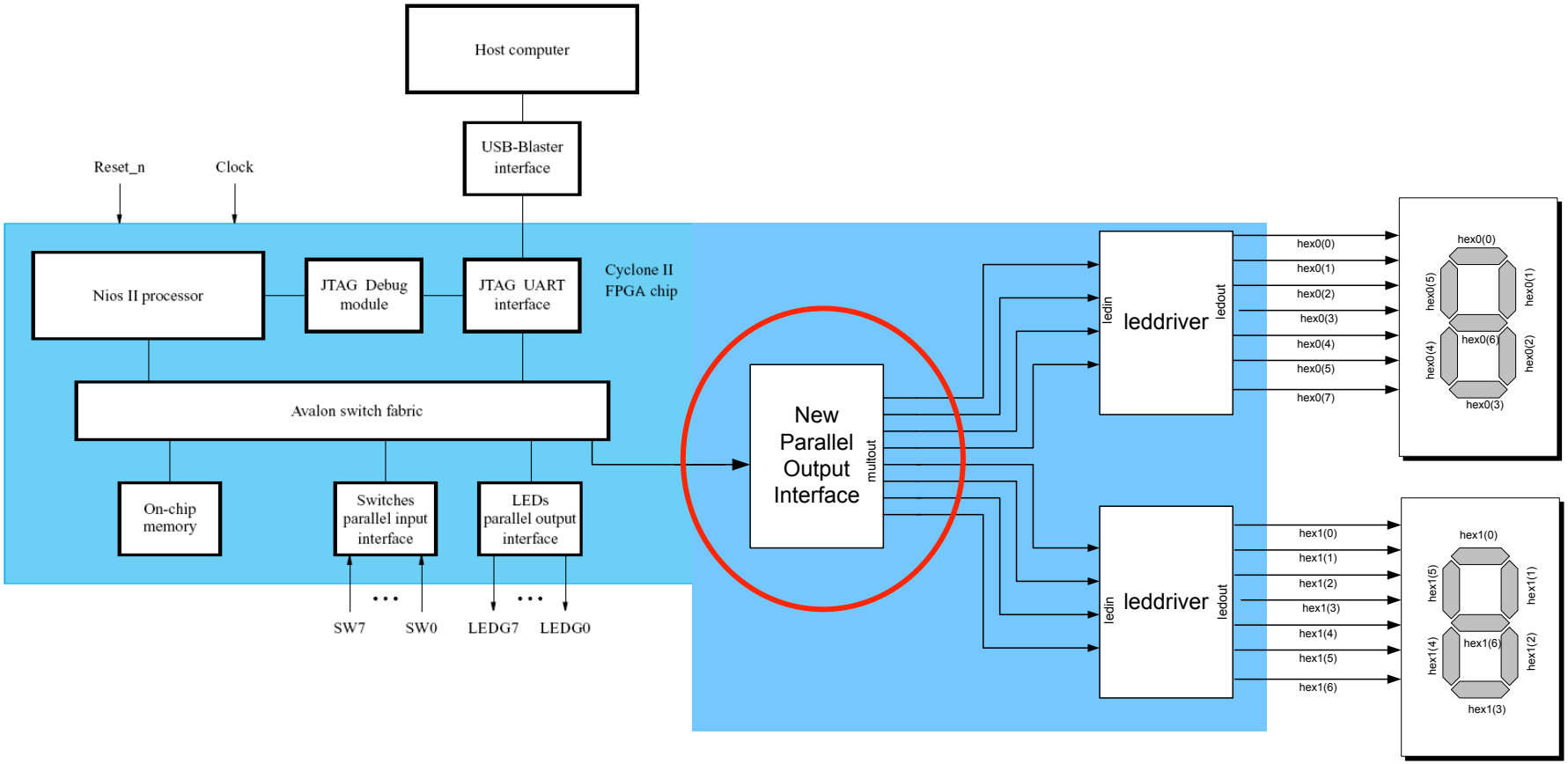
1. Custom Avalon interface



Assign memory address region for this accelerator

Ways of Interfacing a Hardware Core

2. Parallel I/O interface − predesigned by Intel



Ways of Interfacing a Hardware Core

1. Custom Avalon interface

2. Parallel I/O interface

#1 is more flexible, smaller, faster

Introduction to the SoC lab

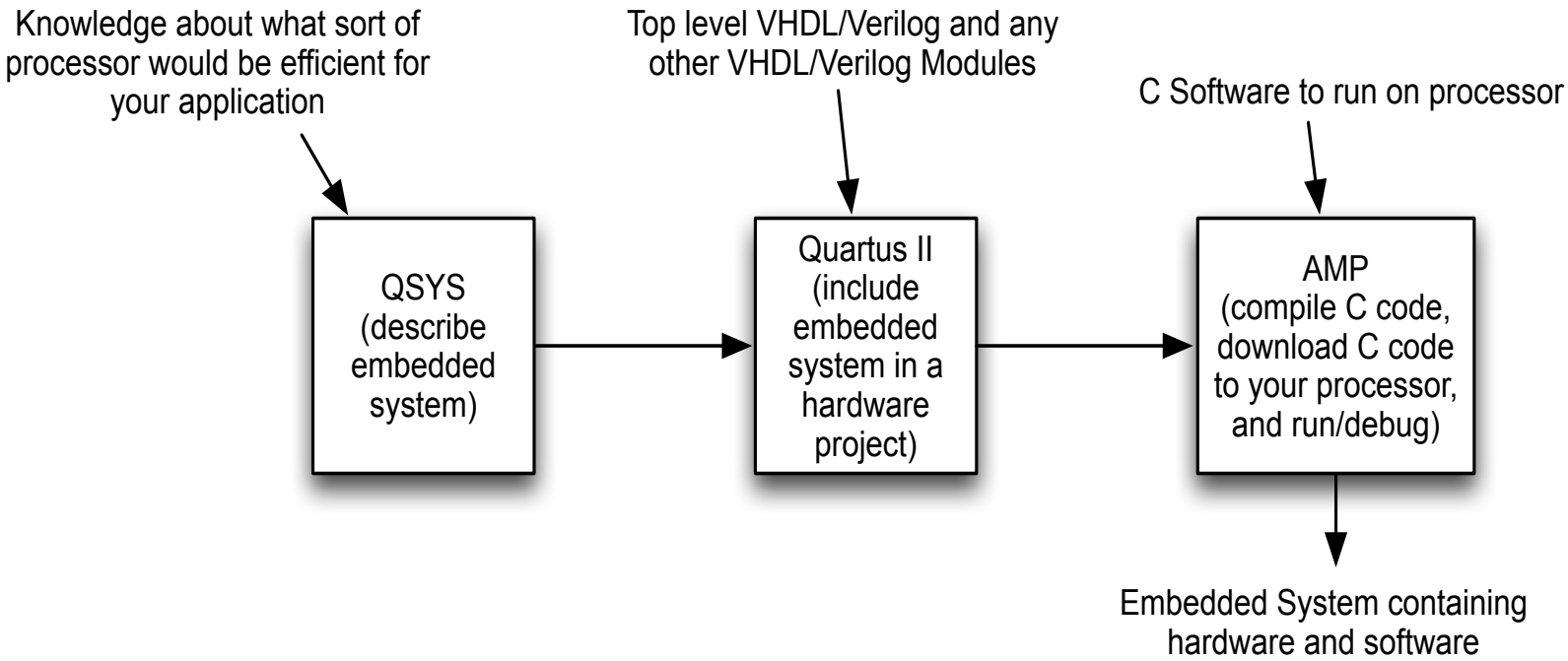
You will get experience with FPGA-supplied tool chains to:

a) Implement and configure processors on an FPGA

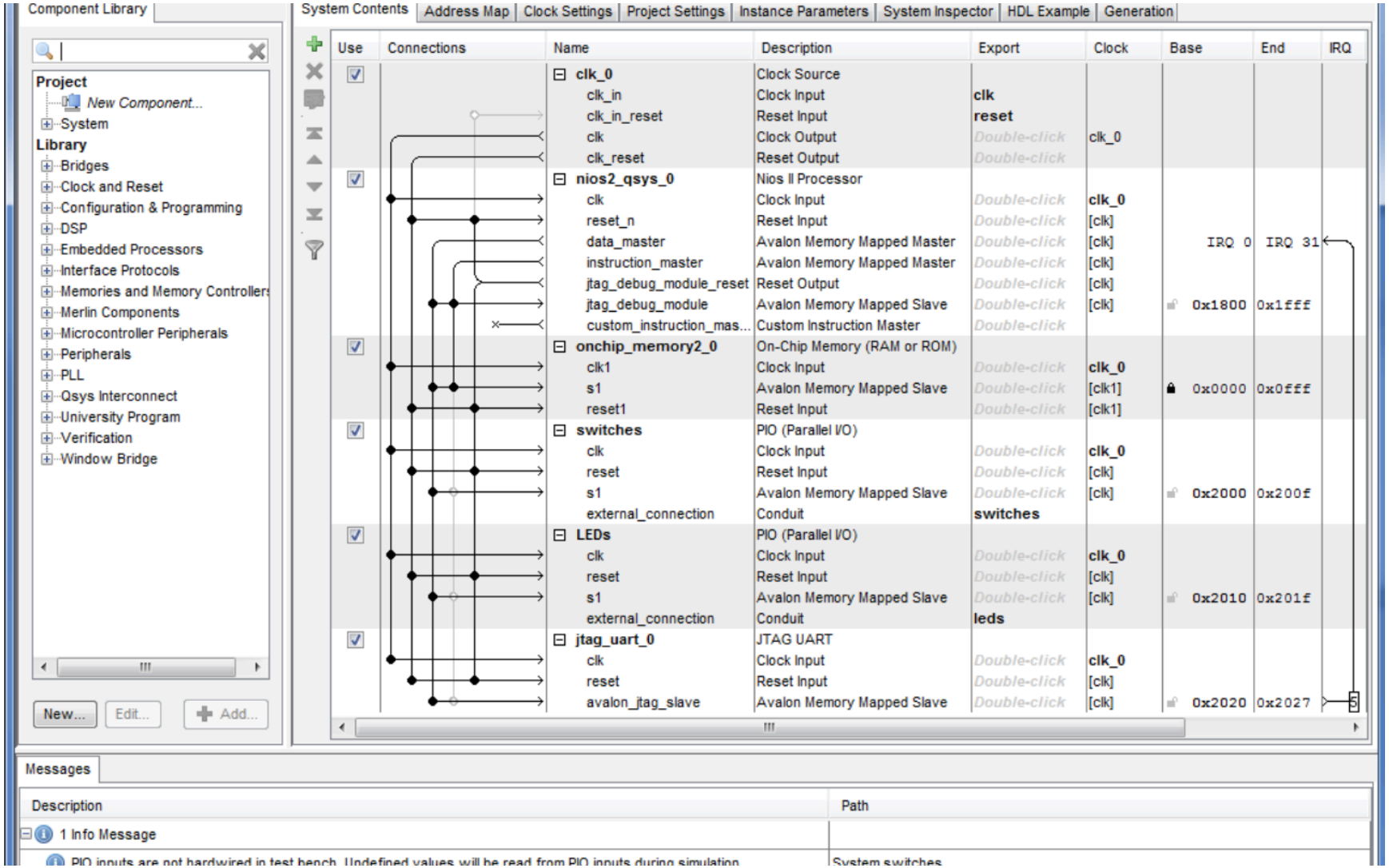
b) Program the processors

c) Program the rest of the fabric (hardware)

d) Interface the hardware and software



Start: Tools->Platform Designer *:

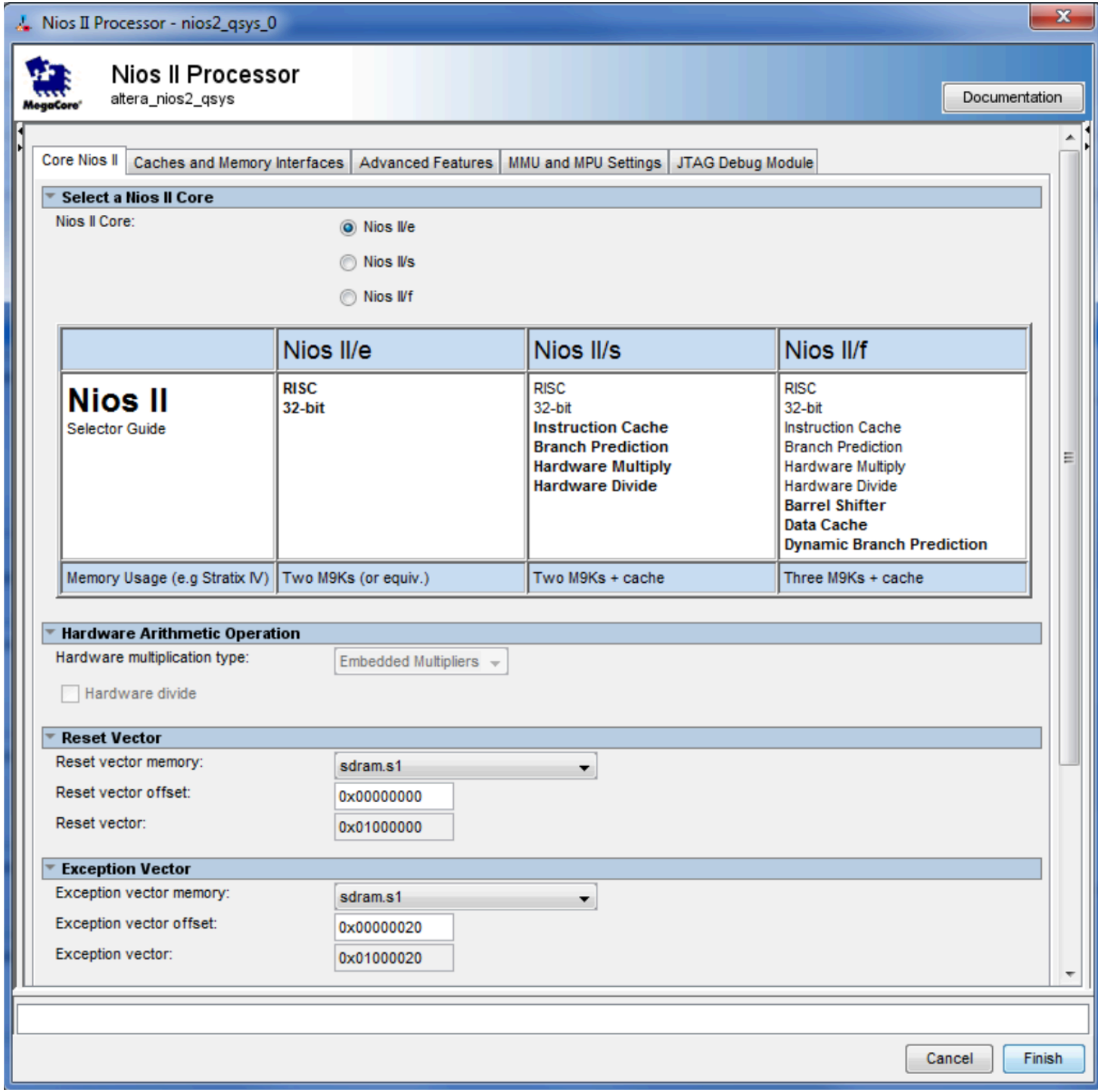


There is a tutorial that describes how to do this, step by step.

* this used to be called Qsys, so you might see references to that name

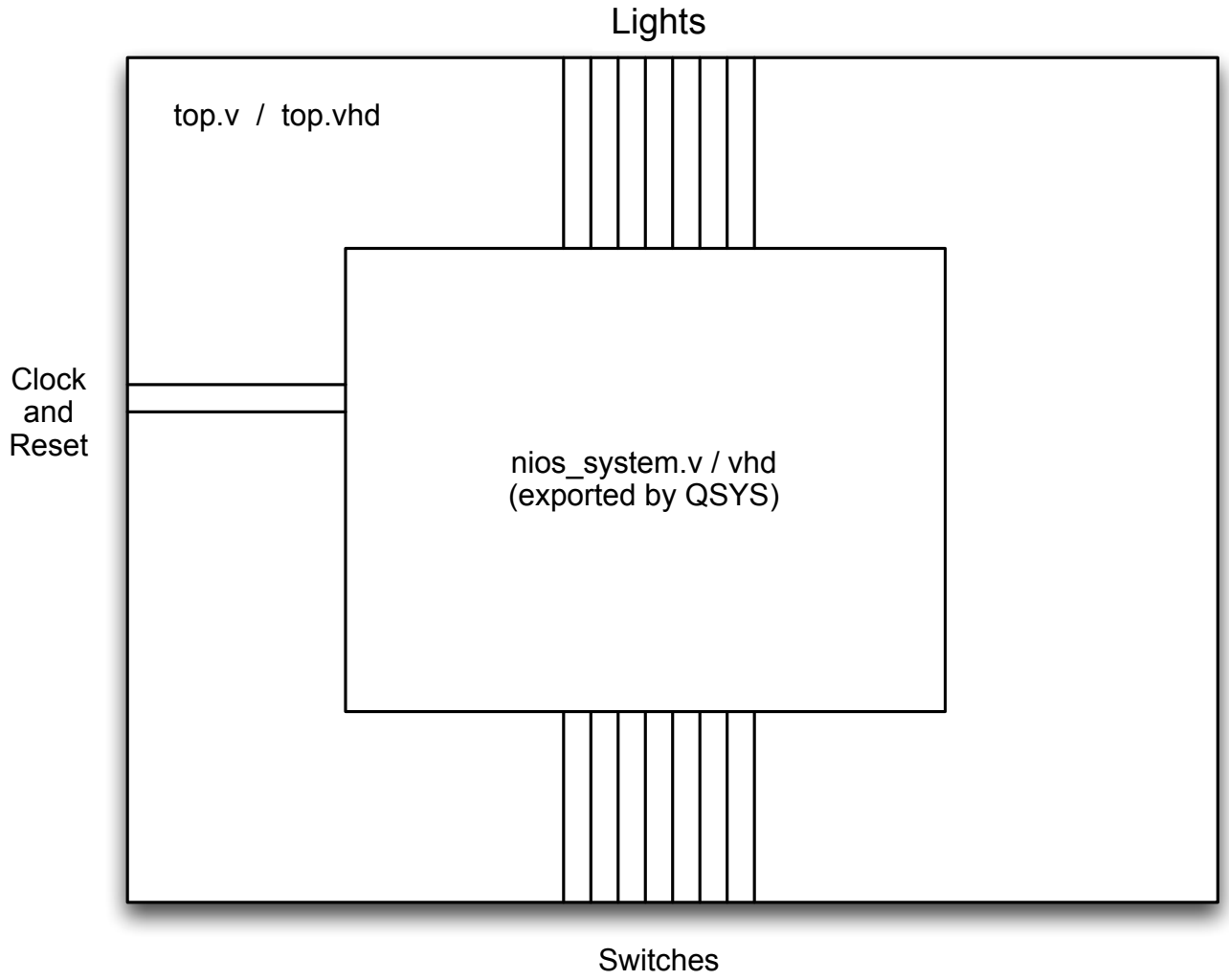
Can configure each unit:

For the processor:



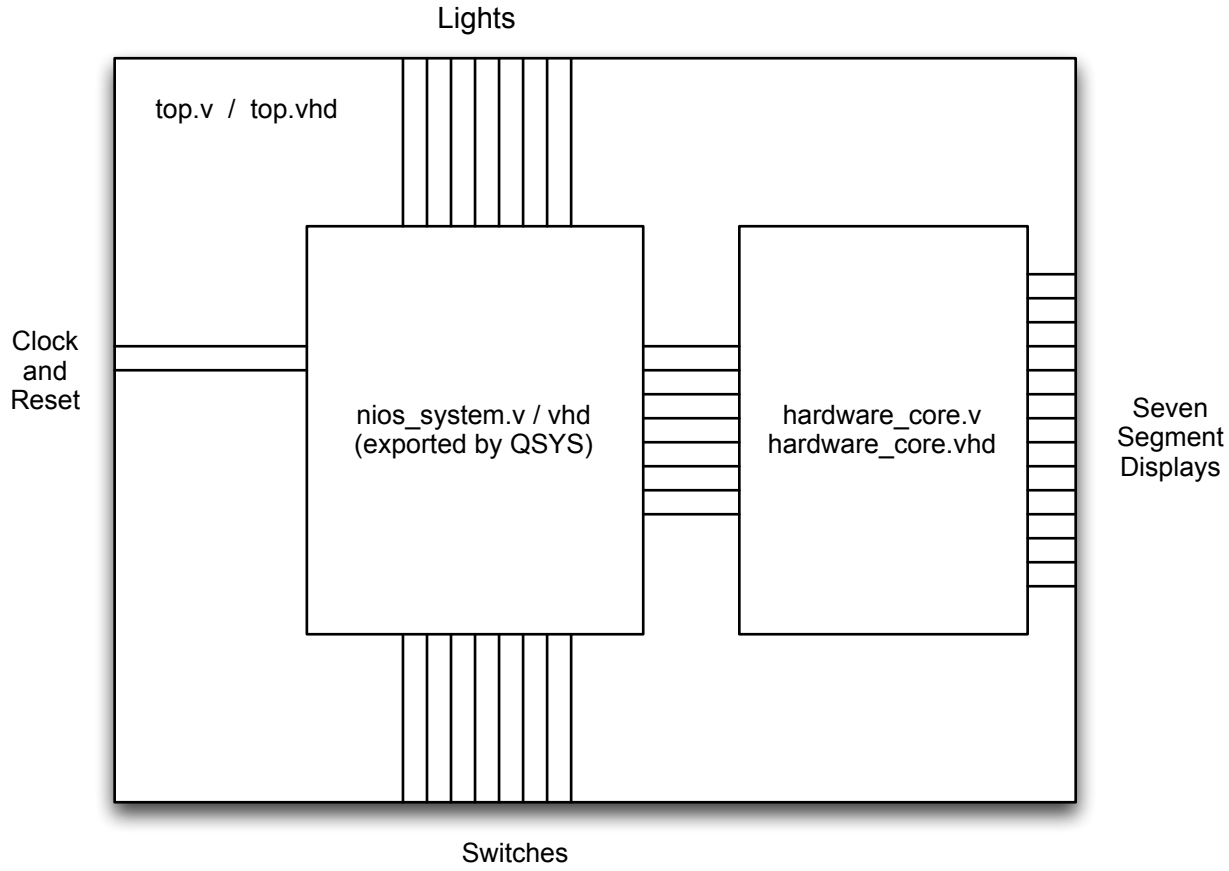
The tool will generate a Verilog description of your system and export it to your Quartus Project.

You will wrap it in a top-level file, ready to compile and download.



Create a custom hardware component and attach it to the Avalon interface

- Use option #1, parallel port interface, to make this easy



Compile the hardware (as normal) and you have an SoC on an FPGA

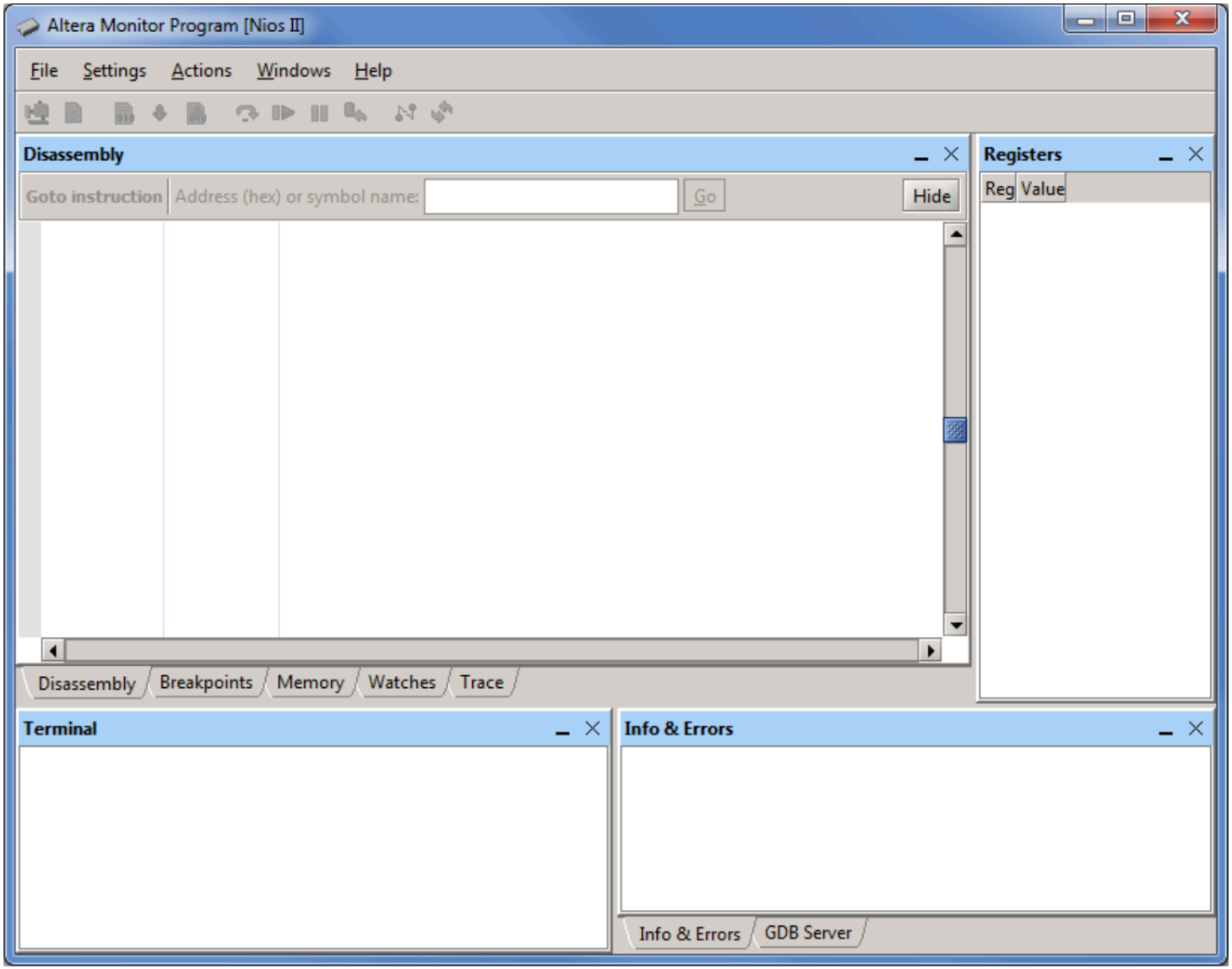
- Processor + peripherals customized for your application

Tools to write and debug software:

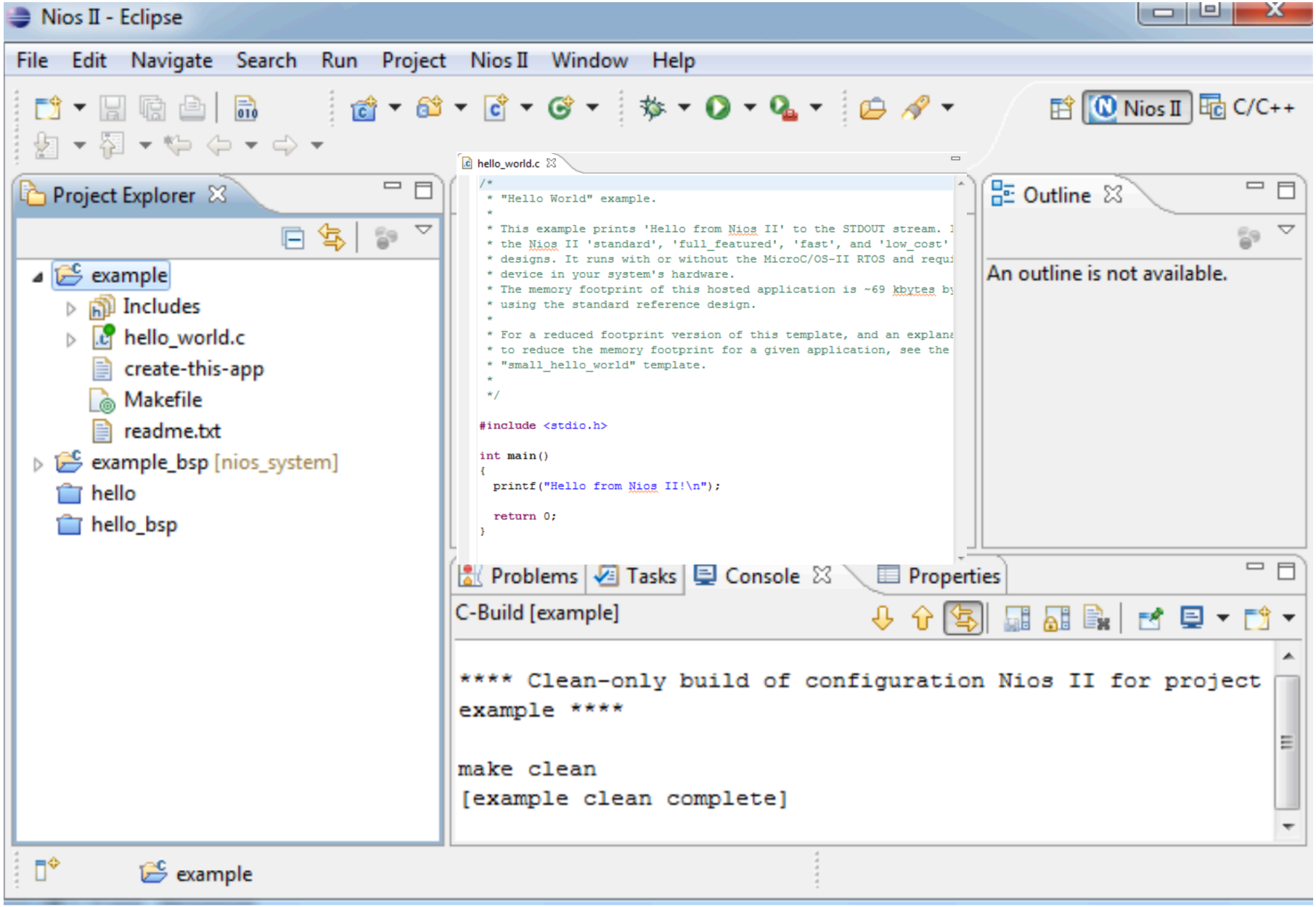
1. Altera Monitor Program (simple)
2. Eclipse (full featured)

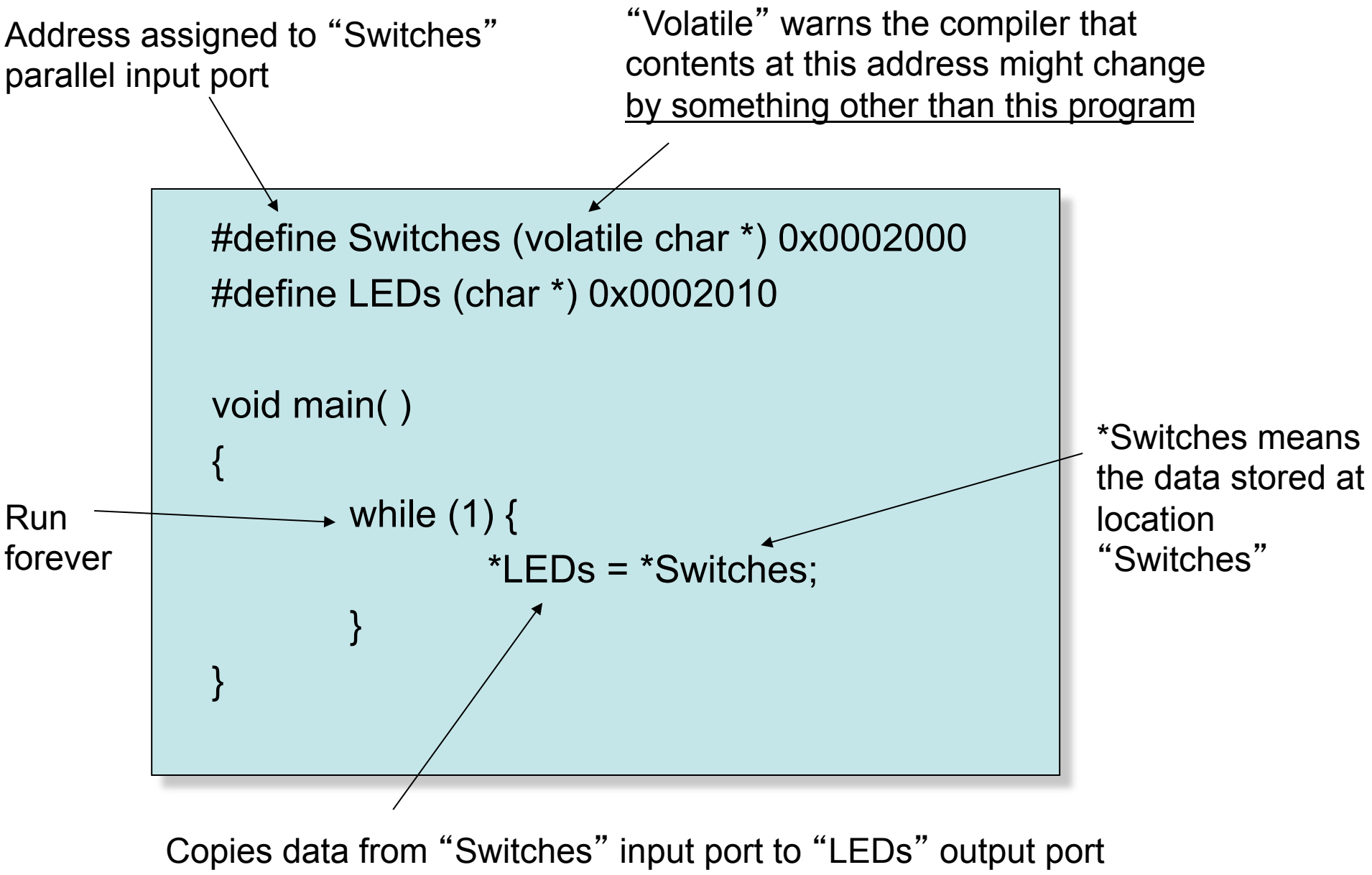
Altera Monitor Program:

Can write code in C or Assembly. We will use C.



Better tool: Eclipse (aka Nios II EDS, Nios II SBT), under Tools menu




Address assigned to “Switches” parallel input port
“Volatile” warns the compiler that contents at this address might change by something other than this program
#define Switches (volatile char *) 0x0002000
#define LEDs (char *) 0x0002010
void main( )
{
while (1) {
*LEDs = *Switches;
}
}
Run forever
*Switches means the data stored at location “Switches”
Copies data from “Switches” input port to “LEDs” output port

This allows you to do all your design in C:

- Writing software. Slow, higher power, all the disadvantages of software

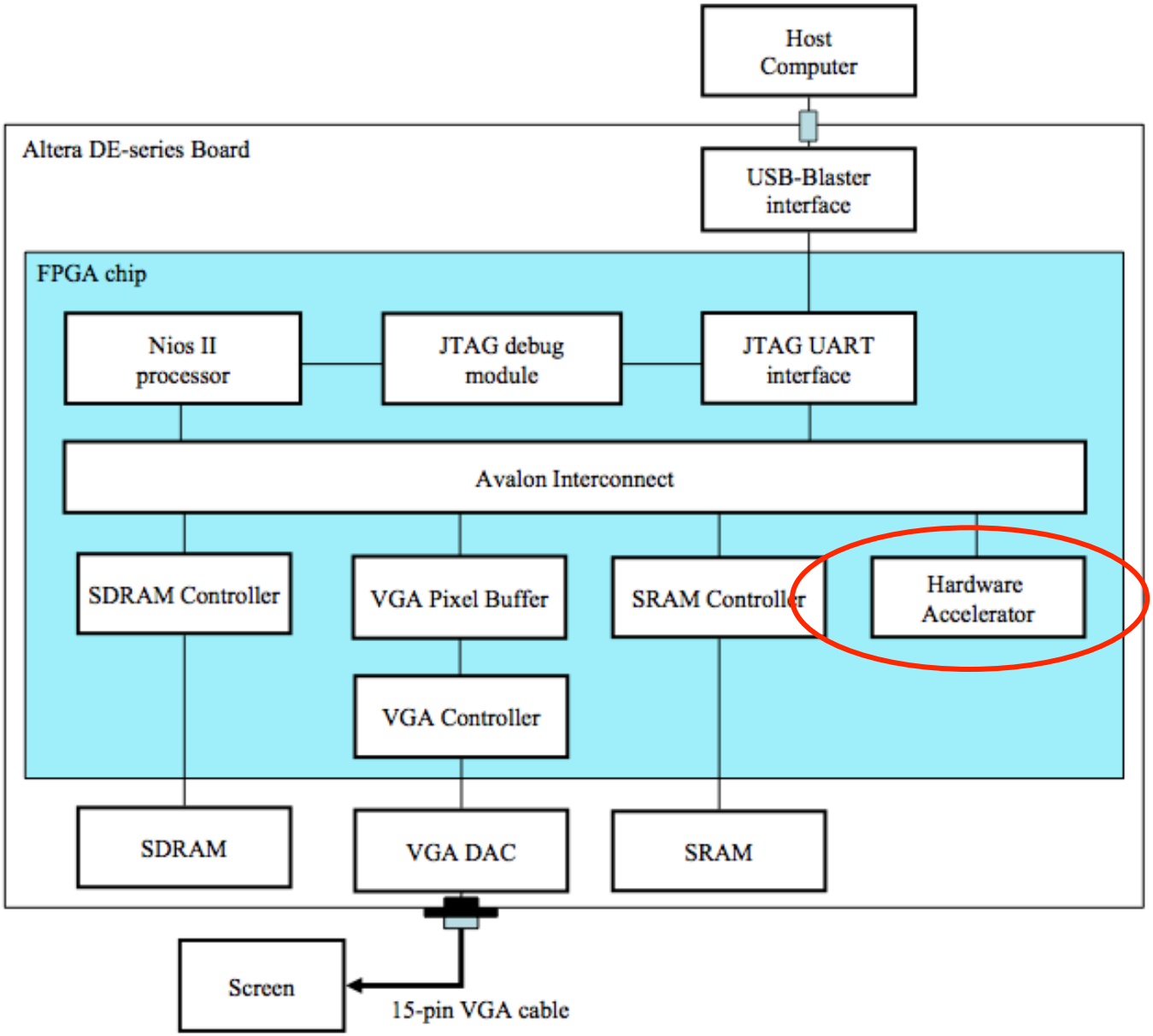
Compared to a stand-alone processor:

- The processor can be customized for your application
- But slower, less than 100MHz

To be really interesting, you want to couple the processor with hardware to accelerate the time-critical parts

Now we will look at the other option for integrating cores:
Attaching it directly to the Avalon bus

Hardware Accelerator



Assign memory address region for this accelerator

Hardware Accelerator

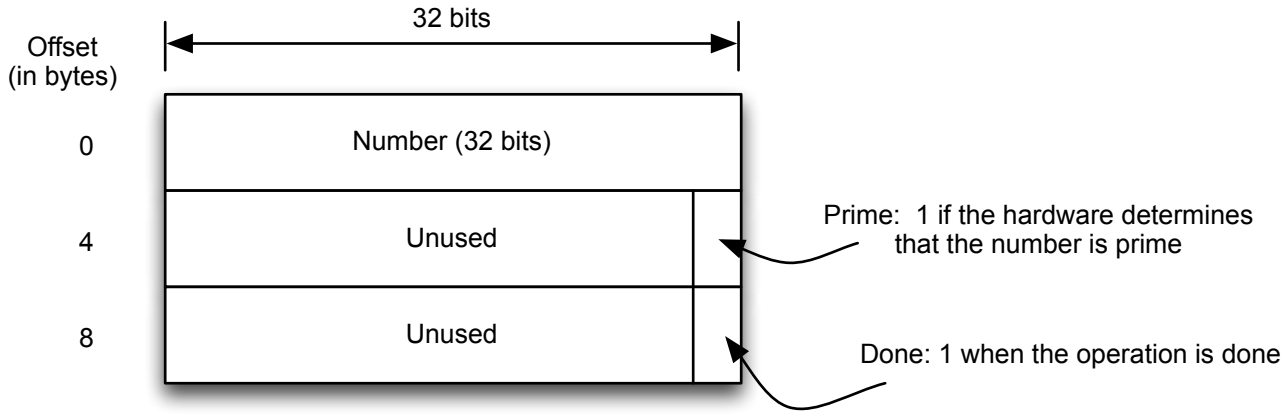
To make this concrete, we will go through two examples:

1. A circuit to determine whether a number is prime (Servant only)
   - Important in encryption like RSA
2. A circuit to draw to a pixel buffer (Master and Servant)

Example 1: Servant

A circuit to determine whether a number is prime.

Step 1: Define Hardware / Software Interface



Software "starts" hardware by writing number to location offset 0.

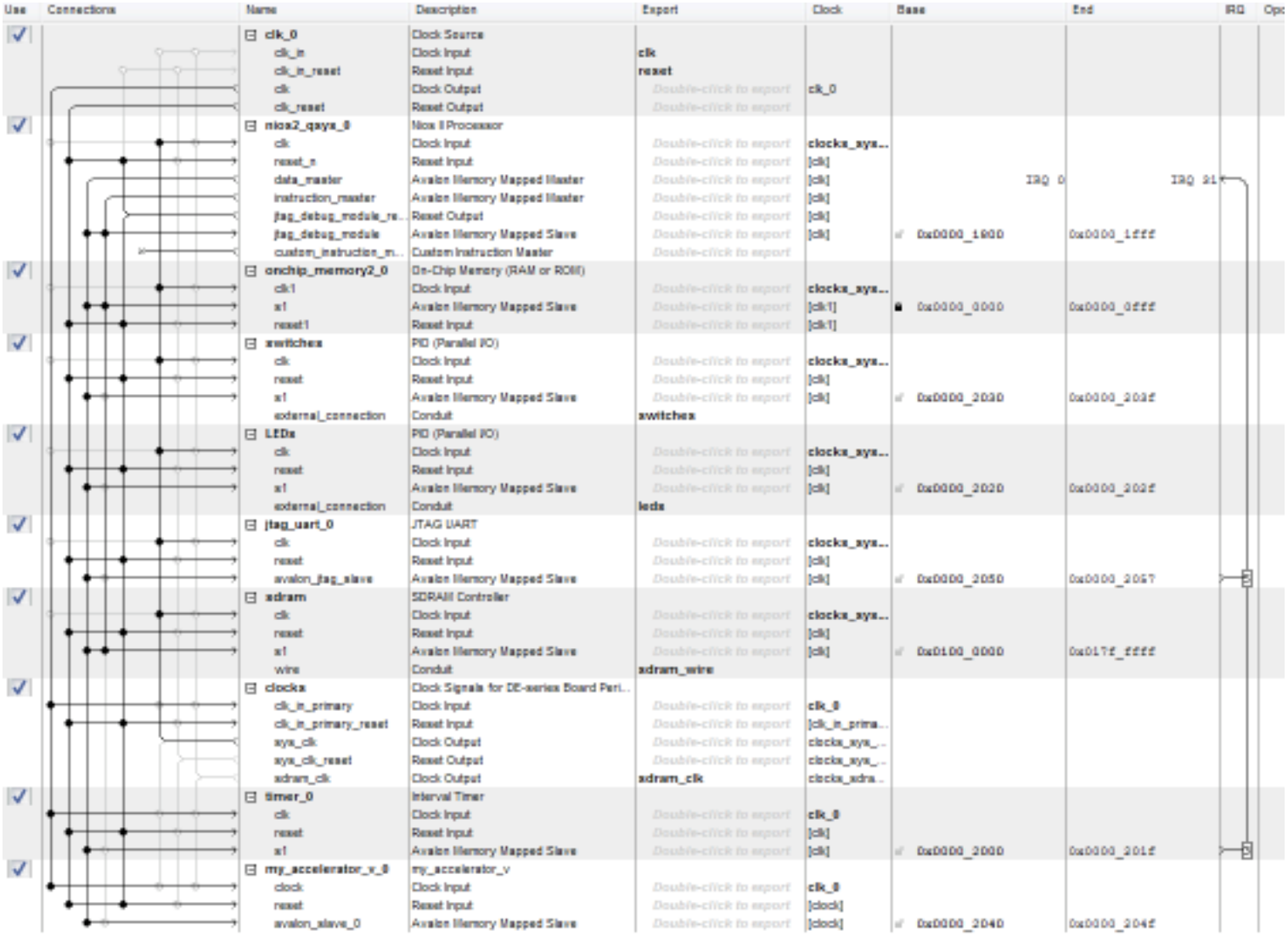
The computation takes multiple cycles.

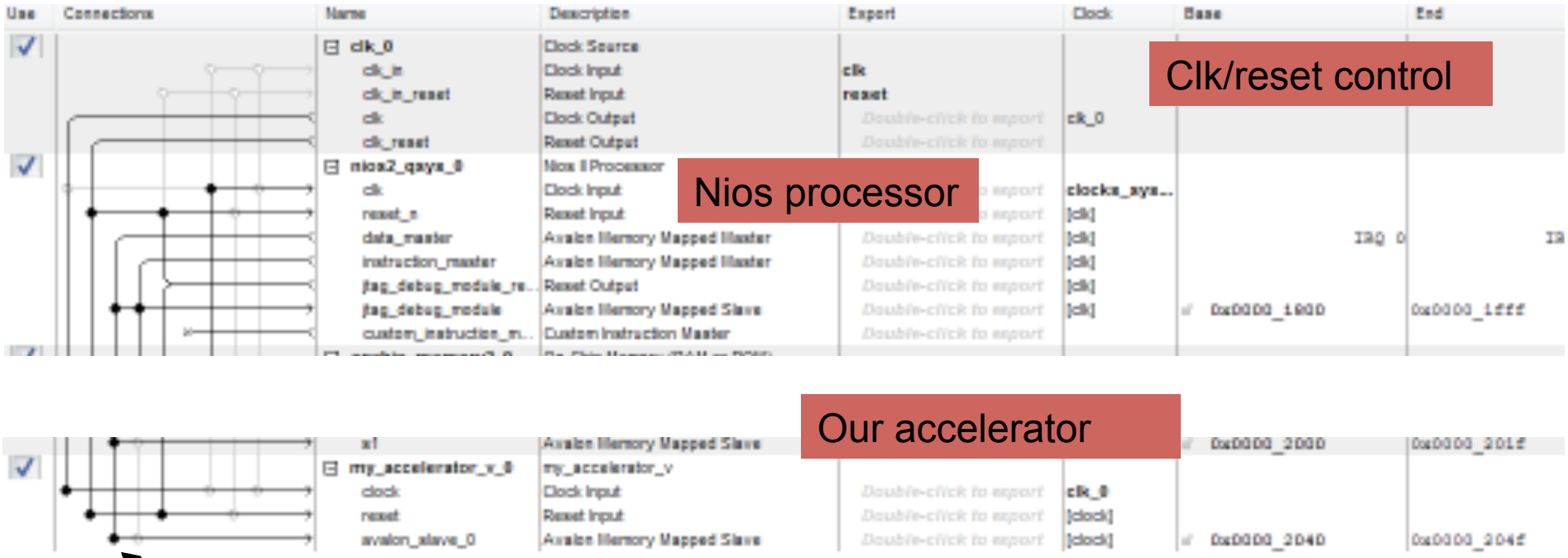
When the computation is finished, bit Done is set to 1, bit Prime gives result.

- Software has to poll, waiting for Done to go from 0 to 1

Other implementations are possible (eg, we could combine Prime and Done into one word)

Example 1: System Design





Connection to Avalon:
processor Data Master bus

Location in memory map that the software can address this core: 0x2040-0x204f

Example 1: Servant

Step 2: Define hardware that makes up the core.

This is a Servant, so hardware is relatively straightforward.

```
// ... reset stuff ...

      // reg declarations not shown

      // See if we are currently processing a request (ie. counting)
      // if not, we can accept a write command from the software

      if (processing == 1'b0) begin

         // we are not processing, so see if the software is trying to
         // write to address 0 (this is the value register)

         if ((wr_en == 1'b1) & (addr == 2'b00)) begin

            // if so, store the value in our internal register
            // and start processing on the next cycle

            saved_value <= writedata;
            processing <= 1'b1;
            done <= 1'b0;
            prime <= 1'b0;
            i <= 2;

         end // if

      end else begin

// ... the actual logic implementation when processing is 1'b1 ...
// CAREFUL: make sure waitrequest is set properly in both cases
```

```
 // This process handles read requests.  It is combinational because
 // we will reply to a read request whenever we see it (don't have to
 // worry about the clock)

 always @(rd_en, addr, saved_value, done, prime, i)
 begin
    if (rd_en == 1'b1) begin

       // see if the location being read is 1 (the count result value)
       if (addr == 2'b01) begin

              // if the user is trying to read the prime flag, only
              // return the count value
              // if done is a 1. Otherwise return a 0

              if (done == 1'b1) begin
                 readdata <= {31'b0, prime};
              end else begin
                 readdata <= 32'b0; // hide prime result until ready
              end // if

// ... other cases here ...
```

Example 1: Servant

Step 3: Write C code:

```
#define my_accel_base (volatile int *) 0x0002040
#define ACC_OFFSET_NUM   0
#define ACC_OFFSET_PRIME 4
#define ACC_OFFSET_DONE  8

#define num 12973   /* a big prime number for testing*/

int isPrime( int my_number )
{
  *(my_accel_base+ACC_OFFSET_NUM) = my_number;  // start

  while( 1 ) {
    if( *(my_accel_base+ACC_OFFSET_DONE) & 1 ) break; // poll until done
  }

  is_prime = *(my_accel_base+ACC_OFFSET_PRIME) & 1; // get result
  return is_prime;
}

int main()
{
  int result = isPrime( num );
}
```

Example 1: Servant

In this example, we get a speedup of between 2x and 1200x over an equivalent software version (depends on the size of the number).

- Software version has to fetch every instruction, lots of overheads

This is extremely optimistic result, though, because I benchmarked it against a very bare-bones processor.

Example 2: Servant and Master

In this example, we will create an accelerator that draws a box in a pixel buffer.

The accelerator must be a Servant:

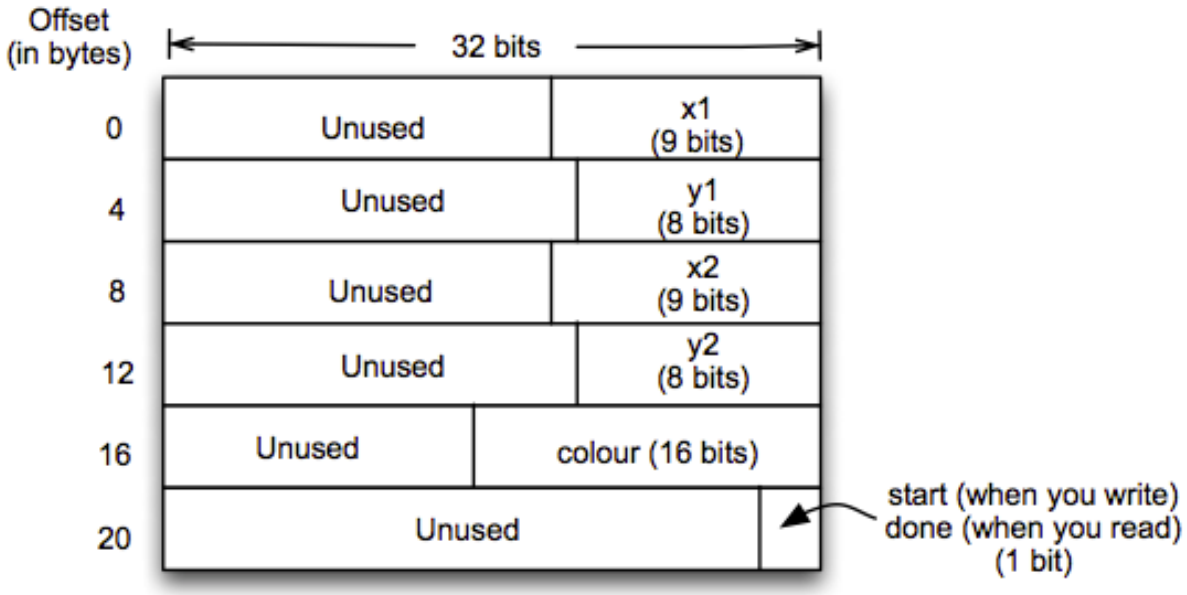
- So the processor can write and read values to the control registers (as in the previous example)

The accelerator must also be a Master:

- The pixel buffer is stored in memory (the memory is also attached to the Avalon bus)
- The accelerator must be able to initiate transfers to write data to the pixel buffer (the individual pixels it wants to turn on).

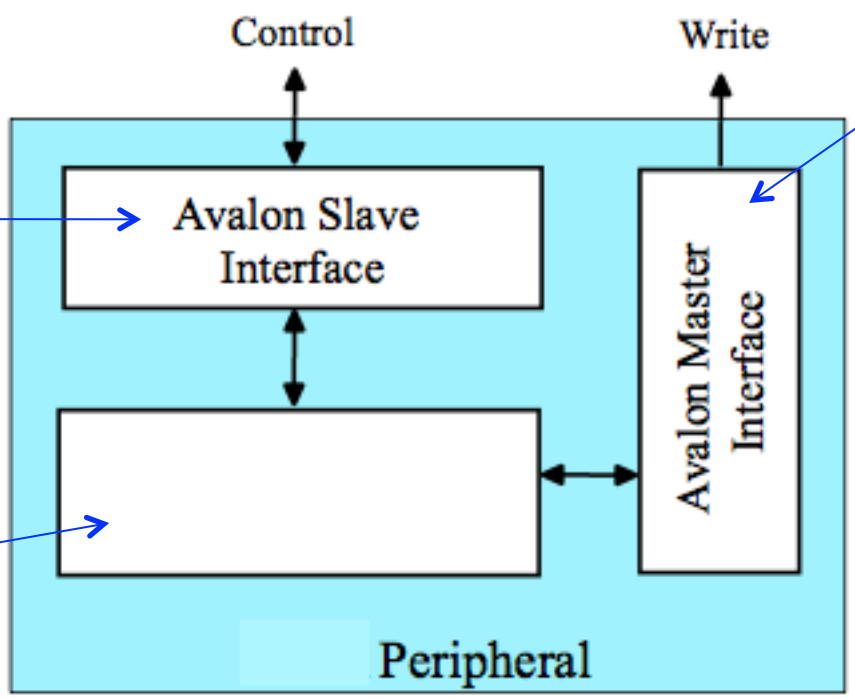
Hardware Acceleration: Example 2

Memory map we will use:



Your Servant interface (which you would write in Verilog) would implement these registers and interface to the Avalon fabric so that the C code can write and read from them.

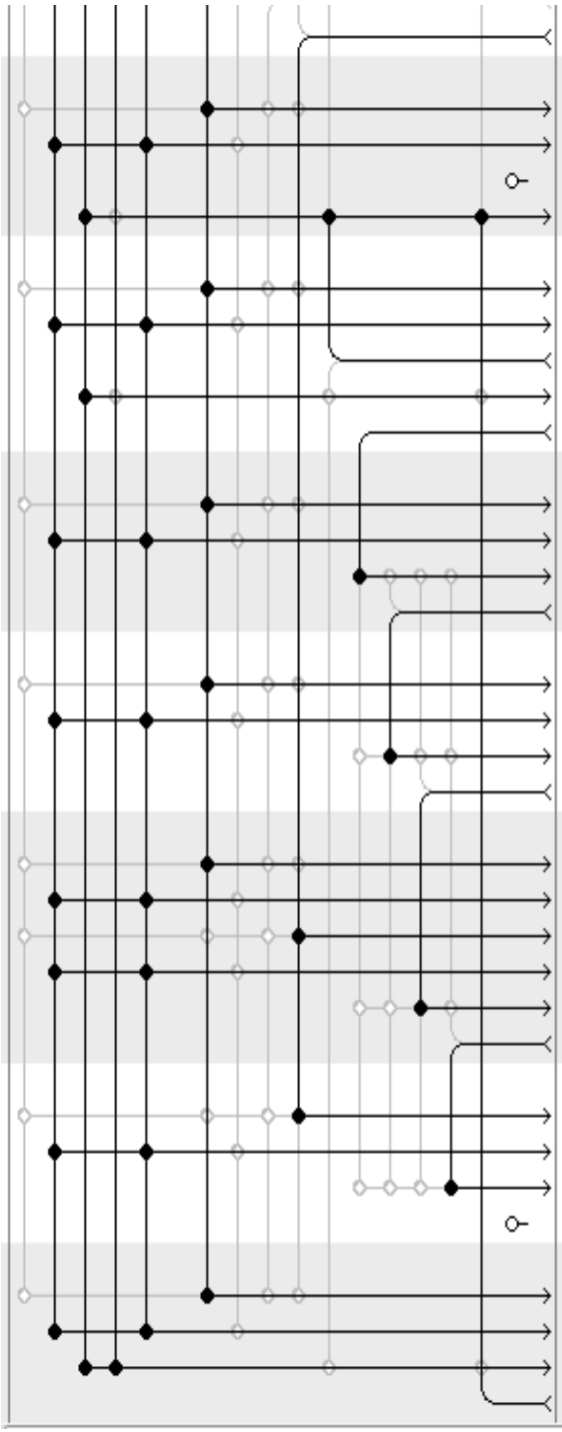
Hardware Accelerator



Used by the processor to control the Accelerator

Calculate points that make up the box

Used by the accelerator to write to the pixel buffer



vga_clk	Clock Output	Double-click to export	up_clocks_0...	
pixel_buffer	SRAM/SSRAM Controller			
clock_reset	Clock Input	Double-click to export	**up_clocks_...**	
clock_reset_reset	Reset Input	Double-click to export	[clock_reset]	
external_interface	Conduit	**sram**		
avalon_sram_slave	Avalon Memory Mapped Slave	Double-click to export	[clock_reset]	0x0008_0000
video_pixel_buffer_d...	Pixel Buffer DMA Controller			
clock_reset	Clock Input	Double-click to export	**up_clocks_...**	
clock_reset_reset	Reset Input	Double-click to export	[clock_reset]	
avalon_pixel_dma_ma...	Avalon Memory Mapped Master	Double-click to export	[clock_reset]	
avalon_control_slave	Avalon Memory Mapped Slave	Double-click to export	[clock_reset]	0x0000_2020
avalon_pixel_source	Avalon Streaming Source	Double-click to export	[clock_reset]	
video_rgb_resample...	RGB Resampler			
clock_reset	Clock Input	Double-click to export	**up_clocks_...**	
clock_reset_reset	Reset Input	Double-click to export	[clock_reset]	
avalon_rgb_sink	Avalon Streaming Sink	Double-click to export	[clock_reset]	
avalon_rgb_source	Avalon Streaming Source	Double-click to export	[clock_reset]	
video_scaler_0	Scaler			
clock_reset	Clock Input	Double-click to export	**up_clocks_...**	
clock_reset_reset	Reset Input	Double-click to export	[clock_reset]	
avalon_scaler_sink	Avalon Streaming Sink	Double-click to export	[clock_reset]	
avalon_scaler_source	Avalon Streaming Source	Double-click to export	[clock_reset]	
video_dual_clock_bu...	Dual-Clock FIFO			
clock_stream_in	Clock Input	Double-click to export	**up_clocks_...**	
clock_stream_in_reset	Reset Input	Double-click to export	[clock_strea...	
clock_stream_out	Clock Input	Double-click to export	**up_clocks_...**	
clock_stream_out_reset	Reset Input	Double-click to export	[clock_strea...	
avalon_dc_buffer_sink	Avalon Streaming Sink	Double-click to export	[clock_strea...	
avalon_dc_buffer_so...	Avalon Streaming Source	Double-click to export	[clock_strea...	
video_vga_controller...	VGA Controller			
clock_reset	Clock Input	Double-click to export	**up_clocks_...**	
clock_reset_reset	Reset Input	Double-click to export	[clock_reset]	
avalon_vga_sink	Avalon Streaming Sink	Double-click to export	[clock_reset]	
external_interface	Conduit	**vga_controller**		
pixel_drawer_0	pixel_drawer			
clock	Clock Input	Double-click to export	**up_clocks_...**	
reset	Reset Input	Double-click to export	[clock]	
avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_2100
avalon_master	Avalon Memory Mapped Master	Double-click to export	[clock]	

Hardware Acceleration

Hardware to actually compute the points that make up the box

- write this in Verilog, not very hard

Hardware to write to the SRAM Controller:

- need to understand SRAM controller timing

Key point: **memory-like interface**

C code to talk to accelerator

Use IOWR to write to registers in accelerator:

```
IOWR_32DIRECT(drawer_base,0,10); // set x1
IOWR_32DIRECT(drawer_base,4,20); // set y1
IOWR_32DIRECT(drawer_base,8,50); // set x2
IOWR_32DIRECT(drawer_base,12,60); // set y2
IOWR_32DIRECT(drawer_base,16,0xFFFF); // set colour
IOWR_32DIRECT(drawer_base,20,1); // start drawing
while( IORD_32DIRECT(drawer_base,20) & 1 == 0) ; // wait until done
```

The last line waits until the box has been drawn.

Demo:

Consider this code:

```
while(1) {
  hw = IORD_8DIRECT(switches_base,0); // get switches value
  if( hw ) {
    IOWR_32DIRECT(drawer_base,0,rand()%160);
    IOWR_32DIRECT(drawer_base,4,rand()%120);
    IOWR_32DIRECT(drawer_base,8,rand()%160);
    IOWR_32DIRECT(drawer_base,12,rand()%120);
    IOWR_32DIRECT(drawer_base,16,rand()&0xFFFF);
    IOWR_32DIRECT(drawer_base,20,1);
    while( IORD_32DIRECT(drawer_base,20) & 1 == 0) /*wait*/;
  } else {
    alt_up_pixel_buffer_dma_draw_box( pixel_buffer,
      rand()%160,rand()%120,rand()%160,rand()%120,rand()&0xFFFF, 0);
  }
}
```

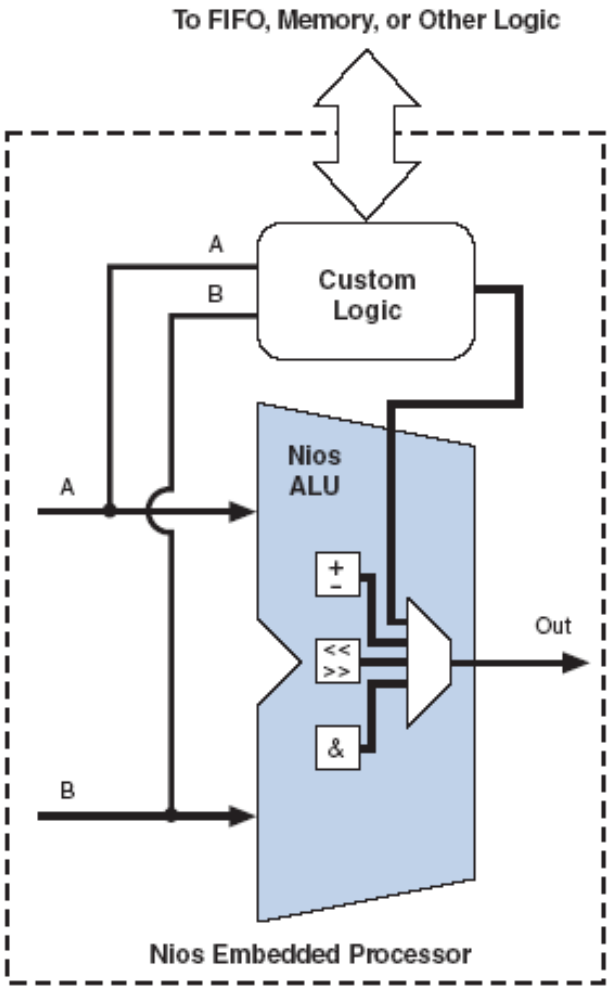
More general form of Hardware Acceleration

You can also use hardware acceleration to compute complex functions quickly:

```
void CcdppCapture(void) {
    CcdCapture();
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] = some complex function;
        }
        bias = (CcdPopPixel() + CcdPopPixel()) / 2;
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] -= bias;
        }
    }
}
```

Another way to do Hardware Acceleration:

You can design “custom instructions” for the processor:



These are assembly instructions, but you can use them in C using macros.

Key observation:

> In many programs, almost all of the execution time happens in a very small number of instructions.

If we can accelerate some of those operations....

Warnings about Hardware Acceleration:

1. Overhead of transferring data:
   - In many cases, it takes more time to transfer the data to the accelerator than it would to execute the original code in software

2. Amdahl's Law:
   - Speedup is limited by the amount of code you can *not* speed up

$$\frac{1}{(1-P)+\frac{P}{S}}$$

P = frac. of execution you can speed up

S = amount you can speed it up

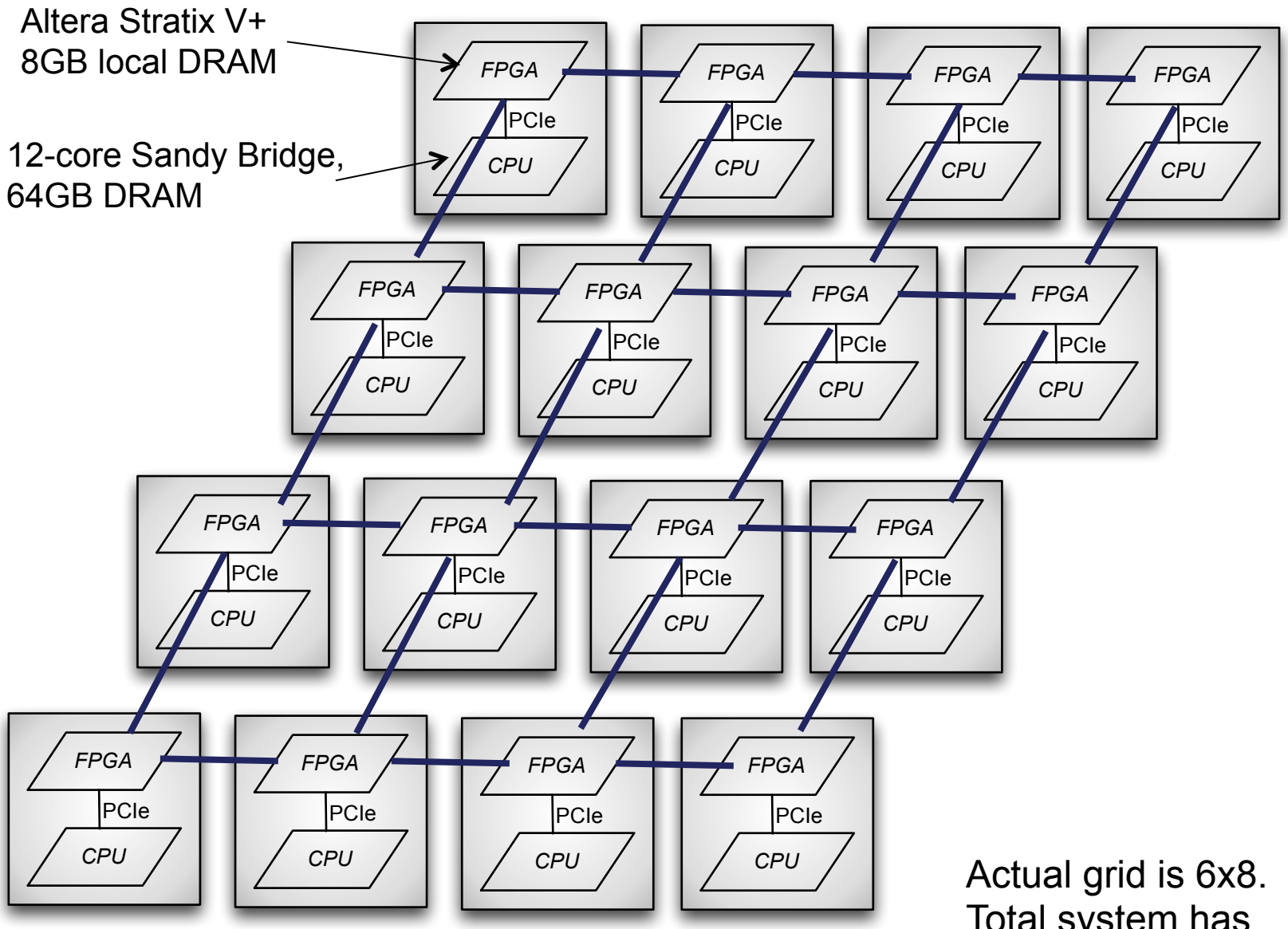
If you can make 25% of your execution run twice as fast, the best you can hope for is a 14% overall speedup.

(Note this is almost certainly not the same as 25% of your source code)

Hardware acceleration is especially interesting for “the cloud”.

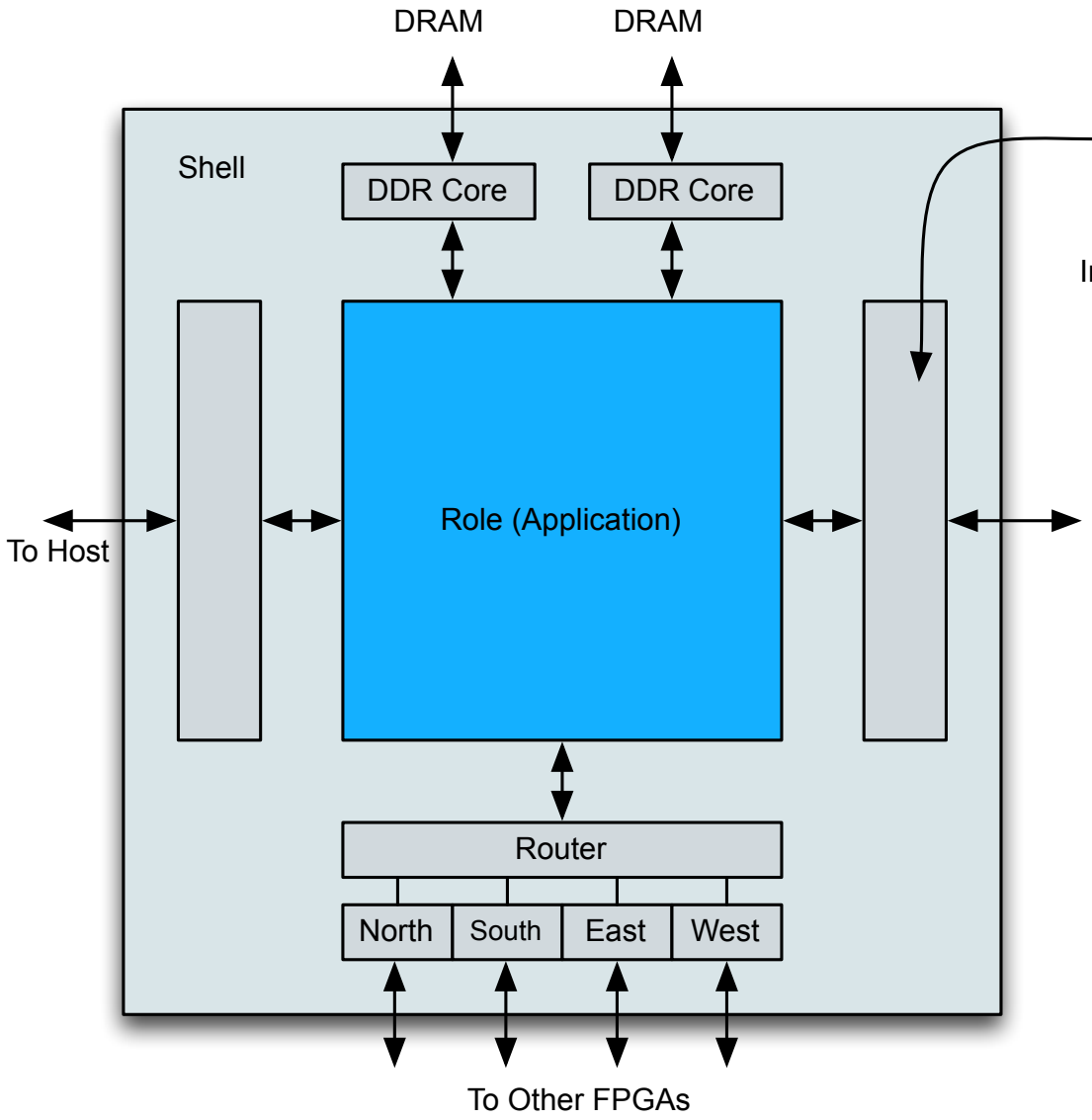
These systems are more complex: multiple processors,
multiple cores, and more complex interconnect

Microsoft Catapult: One FPGA per CPU node



Actual grid is 6x8.
Total system has
34x48 nodes

Catapult FPGA Organization



The FPGA contains a "shell" and "role"

The shell is 23% of chip, leaving 77% for role.

Microsoft has used this to accelerate Bing

"Ranking as a Service": ~ 30,000 + lines of C++ code

Deployed it on 1632 servers. Compared to software only:

- 2x increase in throughput, OR
- 29% latency reduction

Microsoft is continuing to exploit FPGA technology to implement hardware accelerators