Embedded Systems Design with Platform FPGAs

Principles & Practices

Chapter -3 System Design

Dr. Bassam Jamil
Adopted and updated from
Ron Sass and Andrew G. Schmidt

Chapter 3 — System Design

Chapter 3 Learning Objectives

Topics

- principles of system design and **how to assemble systems** on Platform FPGAs
- a general inventory of hardware cores
- review of software resources, conventions, and tools for embedded systems

The evolution of Embedded Systems: from HW to SW

Thirty years ago:

- Embedded systems were primarily custom computing machines built from electronic (discrete and integrated) circuits.
- Most of the work was in the hardware design.
- Software was just some glue.

Now : (SW + Re-usable cores)

- **Commodity hardware** has become very inexpensive.
- Only high-volume products can justify the cost of building custom hardware.
- In many situations, it is best to leverage fast, inexpensive commodity components to provide a semicustom computing platform. In these situations:
  - The final solution relies heavily on software,
  - Software development costs dominate the total product development cost.
  - Despite the fact that software is so easily reproduced and reused from project to project.

Embedded Systems and Platform FPGA

- PFGA provides **a blank slate** and can implement custom computing hardware almost as easily as software.
- While configuring an FPGA is easy, creating the **initial hardware** design (from gates up) is not.
- For this reason, **we do not want to have to design** every hardware project from the **ground up**.
- We want to **leverage existing hardware cores** and design new cores for reuse
- This means we have to know **what cores** are already available, **how to use them**, and **how to build custom hardware cores** that will be (ideally) reusable.

The focus of this chapter

- Discuss the principles of system design. Address the **metrics** of quality design and **concepts** such as:
  - –Abstraction,
  - –Cohesion
  - –Reuse.
- Consider the hardware design aspects, including:
  - – How to leverage existing resources (base systems, components, libraries, and applications).
- Discuss the software aspects of system design. This includes:
  - – Cross-development tools,
  - – Bootloaders,
  - – Root filesystem,
  - – Operating systems for embedded systems.

Outline of This Chapter

3.1 Principles of System Design

- This section focuses on some of the **principles** that guide **good system design**.
- This is far from an exact science and at times it is **very subjective**.
- The best way to read this section is to simply internalize the concepts, observe how they are applied to simple examples, and then consciously make decisions when you are building your own designs.
- In practice, it is difficult to learn good design skills from a textbook. It **comes from experience** and learning from others.
- The goal here is to try to **accelerate that learning** by setting the stage and providing a common vocabulary.

Design Quality

To start we ask: what is “good” and “bad” design? Designs fall into two broad classes:

- **external criteria** characteristics of a design that an end-user can observe
  - pressing volume up button, yet volume decreases
- **internal criteria** characteristics that are inherent in the structure or organization of the design, but not necessarily directly observable by the user
  - coding used in DVR impact ability to fix/maintain design

Clearly, some of these qualities can be measured quantitatively but many are very subjective

Design Quality

Terms relates to system performing **intended function**.

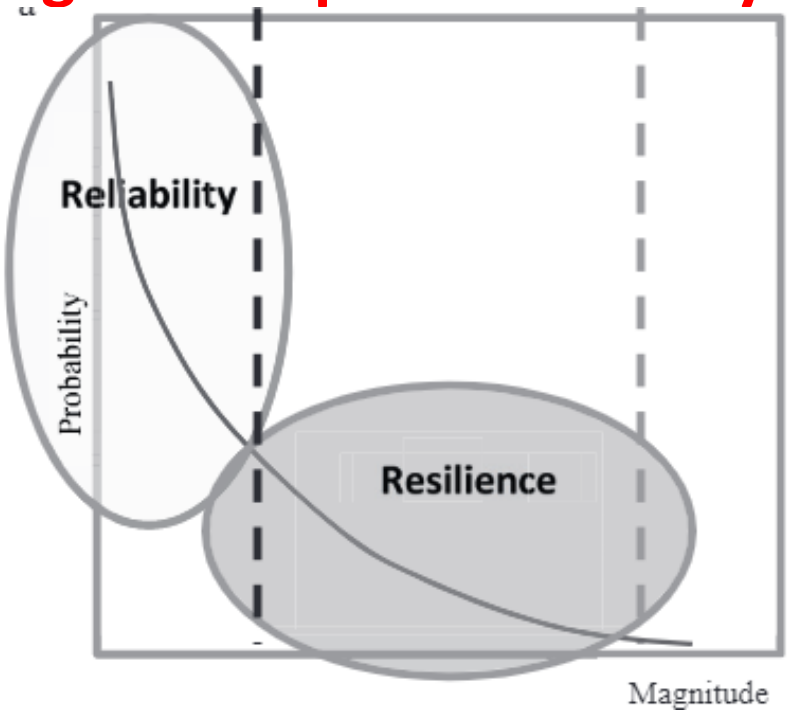
- **correctness** system has been (mathematically) shown to meet a formal specification
- **reliability** means detecting and correcting corruptions.
  - reliable hardware: system behave correctly in presence of physical failures (such as memory corruption due to cosmic radiation)
  - reliable software: system behaves correctly even though the formal specification is incomplete
- **resilience** whereas reliability focuses on detecting and correcting corruptions, resilience accepts the fact that there will be errors and the design “works around” problems even if it means working in a degraded fashion

Design Quality, continue

- Reliability vs. resilience issues:

Reliability issues: **events which may happen during a long time** and their **impacts are not very destructive on the system** .

Resiliency issues : **sudden corruptions which are rare to happen** but their **negative impacts are really strong and destructive** .



Design Quality, continue

- In terms of software,
  - **Reliability** is doing something reasonable even though it wasn't specified.
  - **Resilience** is doing something reasonable even though this should never had happened

- Another term is **Dependability**
  - Dependable system protect the system from a spectrum of issues which spans natural phenomenon to malicious attacks

Correctness Example

For example:

- Embedded systems used in medical systems must not have any human errors in the design.
- Usually accomplished by incorporating additional safety interlocks and **formally proving the correctness** of software-controlled, dangerous components
- With many different interacting components, one would **describe all valid states mathematically and then prove that** for all possible inputs, the software will **never enter an invalid state**.

An invalid state is a state that could harm the patient

Reliability Example

For Example:

- Formally describing all valid states can be **enormously** taxing (and itself error prone).
- Designers may resort to **informal specifications**.
- Informal specifications often **unintentionally omit directions** for some situations.
- i,e. a camera designed to work with USB 1.1, 1.2 and 2.0 plugged into a 3.0 port
- or, an FPGA flying in SPACE is more susceptible to radiation, a more reliable design would be to use Triple Modular Redundancy on critical system hardware and periodically check/update the configuration memory to detect corruption.

Resilience Example

- Resilience and robustness are different from reliability
- Embedded systems interact with the physical world and the physical world is not as orderly as simple discrete zeroes and ones.
- i,e. actuators in machines wear out over time
- **A resilient design behaves correctly even when something that is not supposed to happen, happens.**
- i,e. a thermometer connected to an embedded system may be expected to be in an environment that will never exceed 100◦ Celsius. If, because sensor has become uncalibrated, the sensor begins to report temperatures such as 101, 102, 103, etc. then the system should behave sensibly.
- A reliable system would try to fix the result coming from the sensor; a resilient system might treat 103 as the same as 100 and continue

Other Design Characteristics

- **verifiability** the degree to which parts of the system can be formally verified — i.e., proved correct
- **maintainability** the ability to fix problems that arose from unspecified behaviour
- **repairability** refers to fixing situations where the behavior was specified but the implementation was Incorrect
  A repairable system design allows for easy bug fixes

Other Design Characteristics

- **evolvability** refers to changes that are due to new features
- **portability** refers to a system design that can move to new hardware or software platforms
- **interoperability** refers to a system design that works well with other devices
- terms can be useful during development, discussion, documentation, etc...

3.1.2 Modules and Interfaces

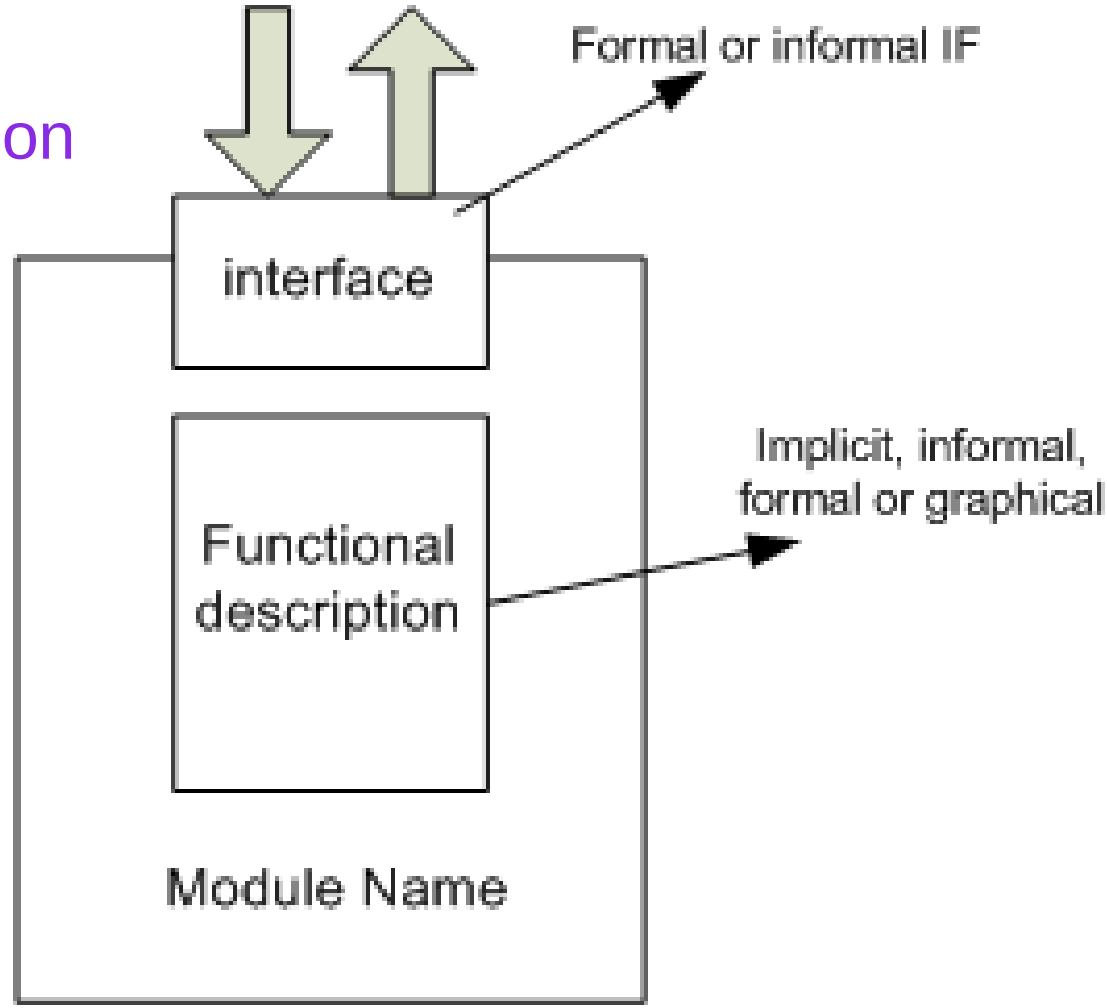
- in order to build complex systems need to first build simple components
- building on these simple components in a **bottom-up** approach
- or, consider the design from the top and work down (**top-down**)

To begin:

- a **module** refers to an self-contained operation that has (1) a name, (2) an interface (to be defined next) and (3) some functional description
- can be hardware, software or a mix of both
- for now easiest to think of as a VHDL component or software subroutine

Module components

Module consists of:
(1) a name,
(2) an interface
(3) some functional description



Modules and Interfaces

Two meanings to term interface:

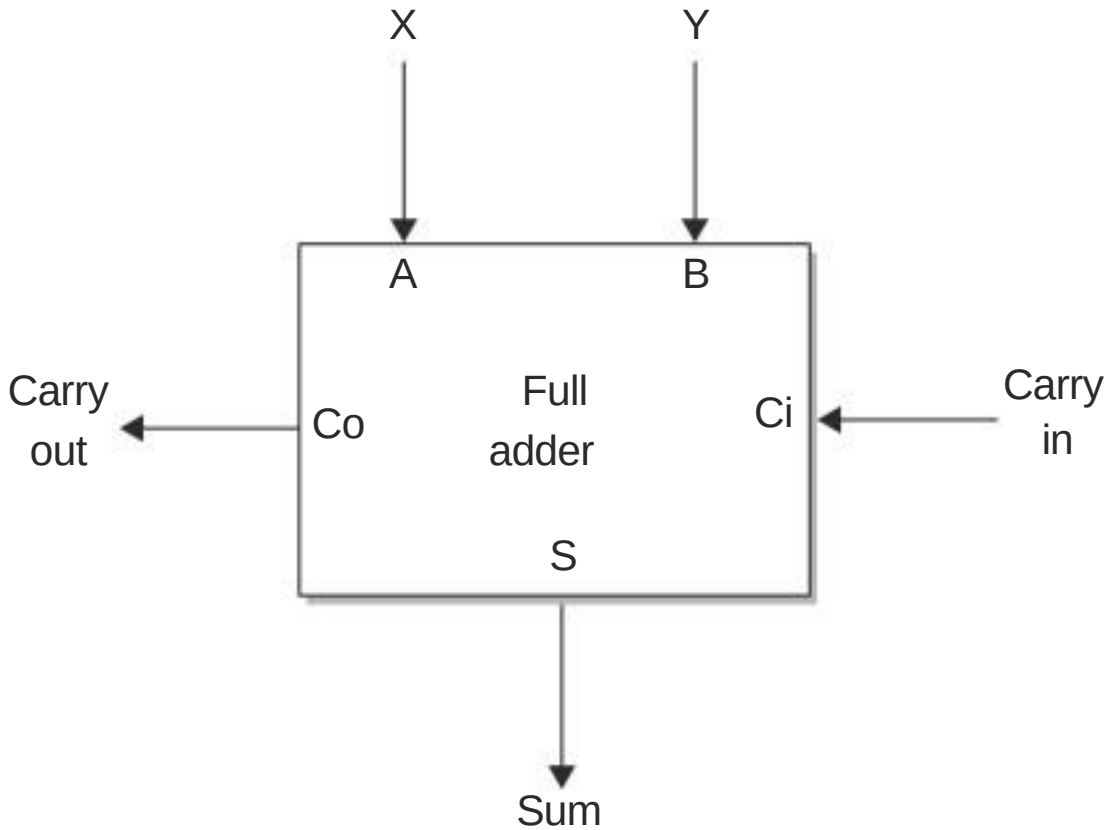
- **formal interface** is the module's name and an enumeration of its operations including, for each operation, its inputs (if any), outputs (one or more), and name
- **general interface** includes the formal interface and any additional protocol or implied communication (through shared, global memory for example)
- formal interface can be mechanically inspected
- i.e, for two modules to interact, their interfaces **must be compatible** and some automated process can check the interfaces
- general interface is more like "how a module is intended to be used" and this cannot, in general, be automatically checked

Modules and Interfaces

- a module can also include a functional description
- the description can be:
- **implicit** the name is so universal that by convention we simply understand what the function is

Implicit Module Example

A module called “Full Adder” we do not need to say any more because the functionality of a full adder is well-known



Informal Module Description

The functional description can be **informal** — its intended behavior is described in comments, exposition, or narrative

Informal Module Description

The functional description can be **informal** — its intended behavior is described in comments, exposition, or narrative

The full adder component will add three bits together, X, Y and a carry in bit. The addition will result in both sum and carry out bits.

Formal Module Description

The functional description can also be **formal** where the behavior is either described mathematically (in terms of sets and functions) or otherwise codified, such as a completed C subroutine or a behavioral description in VHDL

Formal Module Description

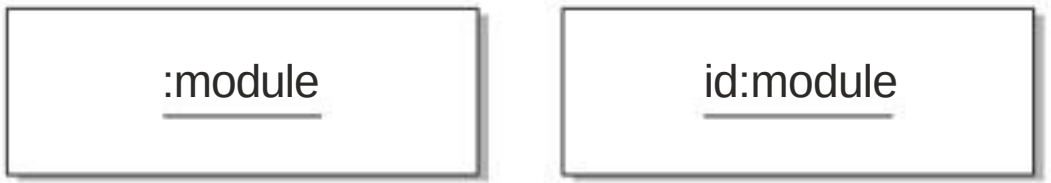
The functional description can also be **formal** where the behavior is either described mathematically (in terms of sets and functions) or otherwise codified, such as a completed C subroutine or a behavioral description in VHDL

```
-- Assign the Sum output signal
S <= A xor B xor Ci ;
-- Assign the Carry Out output signal
Co <= ( A and B ) or (A and Ci ) or (B and Ci );
```

Graphical Module Depiction

- graphically, a module is very simple to denote
- it can be as simple as just a box
- or, more specifically, we can give a module a name
- need to distinguish between a module versus an instance
- instances are shown with the module name underlined
- can name instances too in the event multiple exist

Instance Name



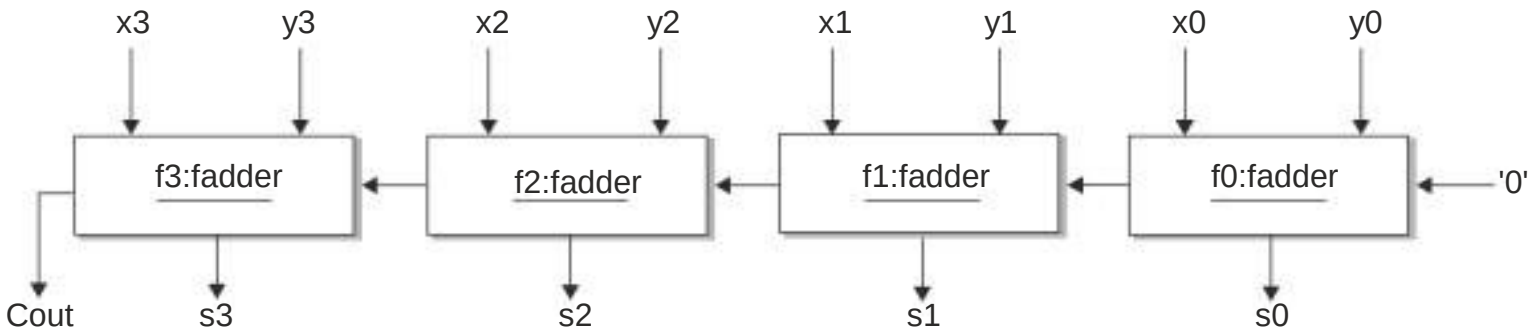
(a) (b)

two instances of a module system, (a) the default instance format and (b) the id to give the instance a unique name

Implementations and Instances

- **implementation** (of a module) is some realization of the module's intended functionality
- **instance** is a use of an implementation
- in software, there is generally a one-to-one relationship between an implementation and instance because the same instance is reused in time
- in hardware, it is common to use multiple copies of an implementation; each copy is an instance

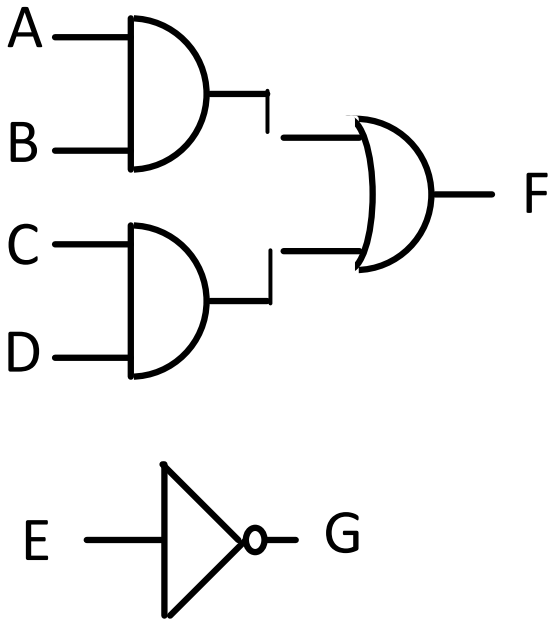
Implementations and Instances



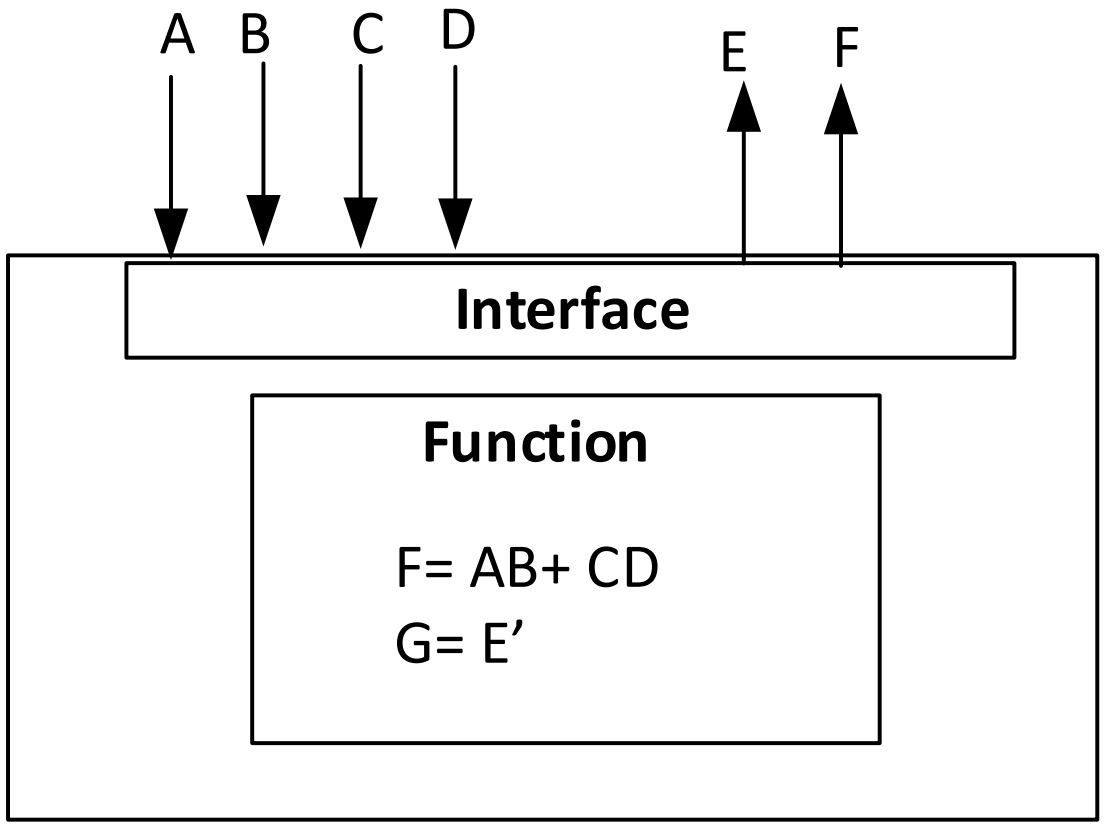
four formally defined modules to generate a 4-bit full adder from 1-bit full adders

Formal Module Example

Design



Module



3.1.3. Abstraction and State

- two major concepts in system design:
  - abstraction
  - state
- apply concepts to modules described earlier

Abstraction

- **abstraction** is the act or an instance of abstracting or taking away; an abstract or visionary idea
- hence, **a module is an abstraction** of some functionality in a system
- what makes for a good abstraction?
  - interface and description provides some easily understood idea
  - details of implementation are more complex
  - i.e, **capturing all salient features** of an idea and **cuts out low-level details**

Importance of Abstraction

- The goal of good abstraction: hide the details that do not serve a purpose.
- The objective of creating abstractions is to overcome the fact that humans can only keep a relatively few number of things in our short-term memory.
- Example of abstraction: the map of the London subway.
  - First published in 1908. The map was rich in detail.
  - In 1933, the London Underground map was changed.
  - By abstracting away much of the detail, the map could print the station names in a larger font.

State

- from a hardware designer's perspective:
  - consider sequential "state" machines
  - can point to the memory devices (where state is stored)
- from a system designer's perspective — more complicated
- state can be stored in many different places (devices)

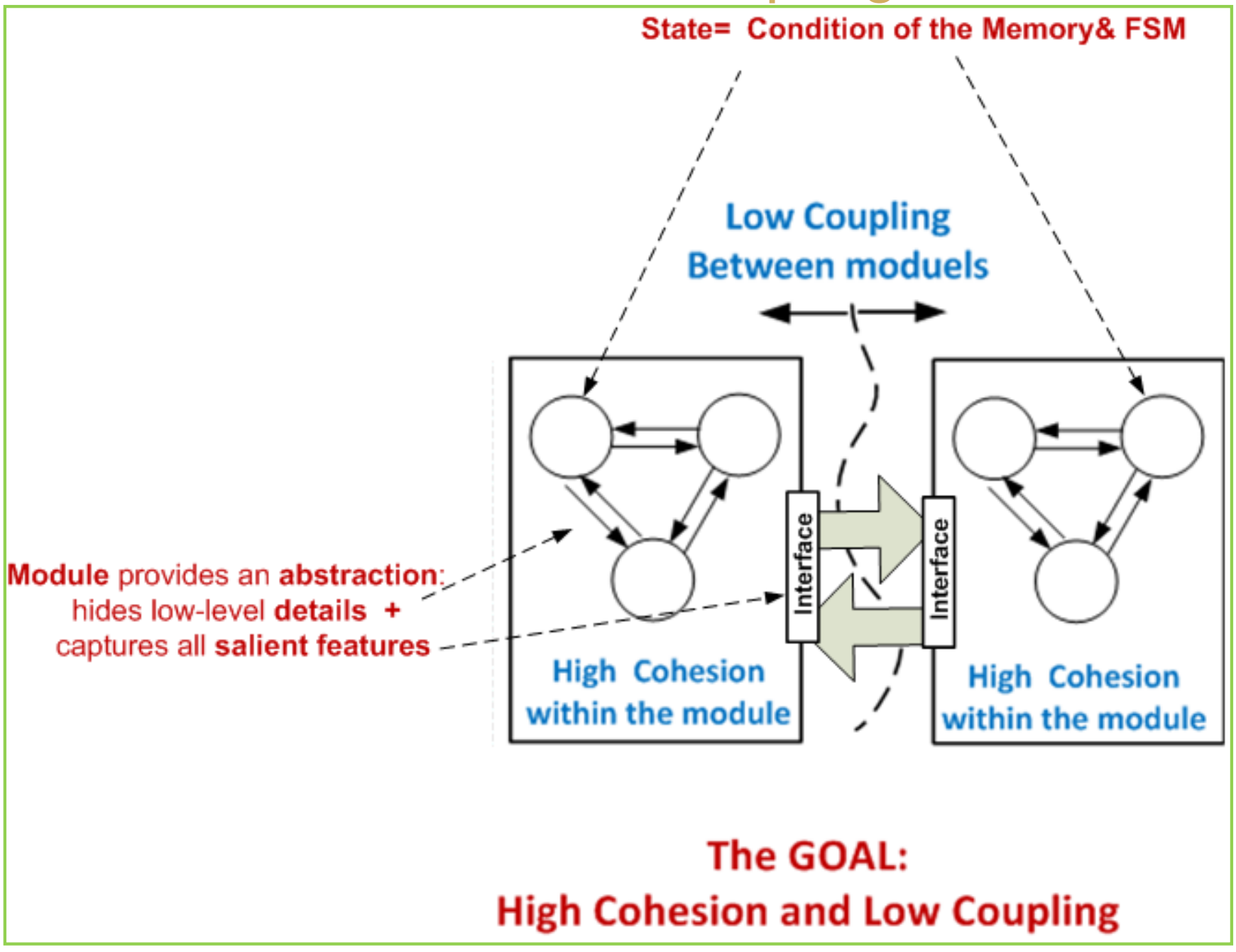
State

- **state** is a condition of memory
- something “has state” or is “stateful” if it has the **ability to hold information over time**.
- identifying the state in a module is important in system design
- In system design, we need to separate functionalities into modules . Abstraction and state are major factor on deriving modules.
- Good abstraction and careful management of state will lead to good modules and improve the design
- The designer needs identify the states a module can be in and what operations might change that state.

3.1.4. Cohesion and Coupling

- Cohesion and coupling are measures (i.e., metrics) to measure abstraction and state.
- Cohesion:
  - Measures abstraction,
  - A model has cohesion if **details** inside of a module come **together** to implement an **easily understood function**.
- Coupling:
  - Measures the state,
  - A measure of how **modules** of a system are **related** to one another,
  - A system's coupling is judged by the **number of and types of dependencies** between modules.
  - High degree of coupling have this cascading effect where a simple change cannot be made in isolation.
  - Coupling forces the designer to consider the whole module and understand everything in order to ensure that a change does not break something.
- **The GOAL: High Cohesion and Low Coupling**

Cohesion and Coupling



Types of Dependencies

- Definition of Dependence :
  - "if a change to module A requires a designer to inspect module B (to see if the change impacts B), then B depends on A.
- Explicit dependency :
  - For example, if the output signal of module A is the input to another module B, then we say that A and B depend on each other.
- 1. Non-explicit dependency : "Sharing the state"
  - If two modules share state, then there is dependence.
  - For example, if one module uses a large table in a Block RAM to keep track of some number of events and another module will occasionally inspect that table, the latter is dependent of the former.
  - If someone wants to make a change to the format of the table, that change will impact the latter module.
  - This is where explicitly identifying state becomes critical to the formation of modules within a system

Types of Dependencies, continue

- 2. Non-explicit dependency : “Timing dependence”
  - For example, they may be dependent because the system may rely on them completing their tasks at the same time.
  - Hence, the system is coupled in time.
- 3. Non-explicit dependency : “system-level dependence”
  - Another example might be in an embedded system that has sensors and actuators.
  - Two modules may not communicate directly, but if one module is changing an actuator that another module senses, there may be a dependence.

Dependency and Good System Design

- Dependence in itself is not bad.
  - Some dependence is necessary between modules because the modules are designed to work together to form the system.
- Dependency is judge by the **degree of coupling** in the system — that is, **the number and type of dependencies**.
- Good dependency is based on :
  - Explicit dependencies based on formal interfaces,
  - Modules with a unidirectional sequence of dependencies.
- Bad dependency results from :
  - Many implicit dependencies,
  - Circular dependencies (where one module A depends on module B and B depends on module A),
  - large numbers of dependencies

I. Reducing Coupling By Encapsulation

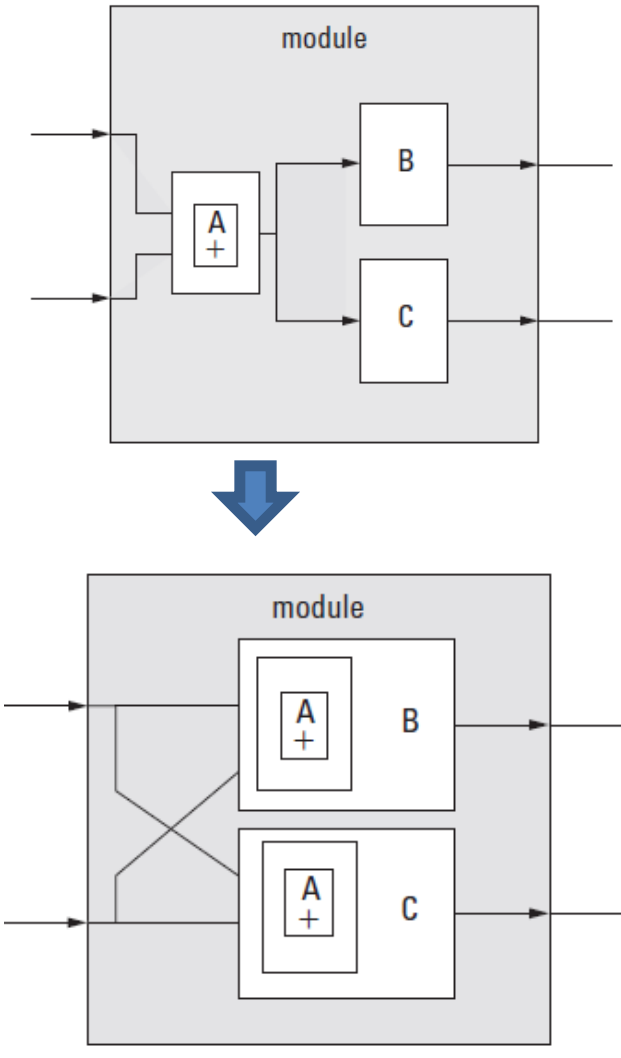
- Encapsulation is often called “information hiding,”
- Encapsulation involves moving state into a module and make it exclusive (not shared). It manipulates state and introducing formal interfaces
  - Result: if one wants to change the module, then there is much more freedom to do things like change the format of the state without the risk of introducing a bug into another module.

Sounds like Object Oriented Programming !!!

- If the module has good abstraction, then information hiding also allows the module be reimplemented in isolation.
  - You can change internal design of a module as needed.
  - All that is necessary is to keep the interface the same

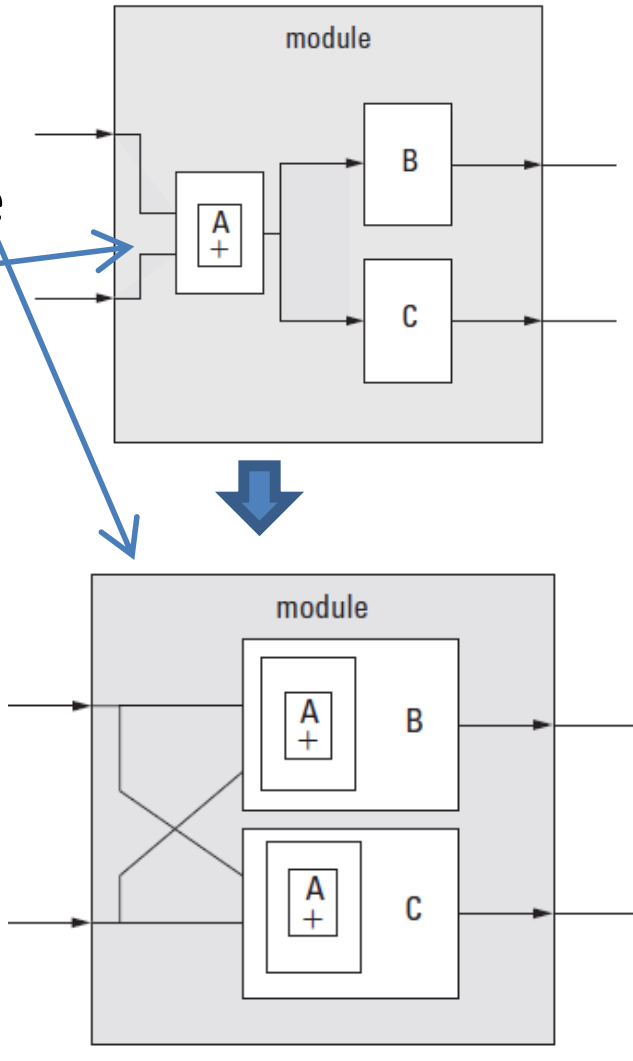
II. Reducing coupling : Reduce Degree of Coupling

- We can decrease the degree of coupling by **duplicating modules**.
  - The goal is to avoid coupling when it is unnecessary.
- Consider the shown example.
  - In the first design, we had two dependencies — submodule B depends on submodule A and submodule C depends on submodule A.
    - x and y, are summed together in submodule A and the results are passed to the submodules B and C
  - In the second design there are no submodule dependencies, so we have clearly reduced the coupling in the design.



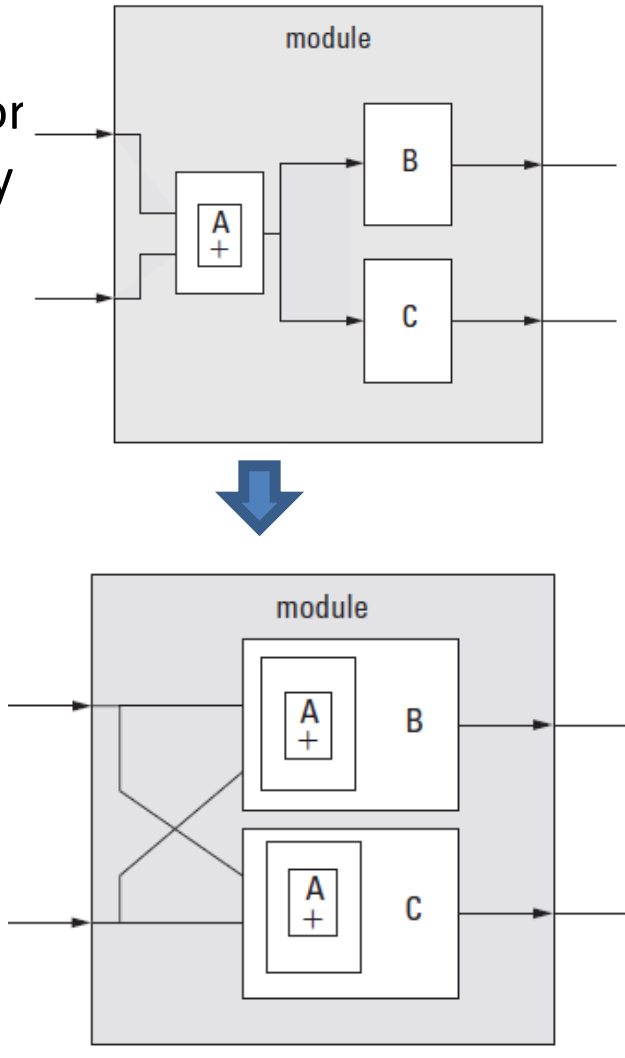
Reducing coupling : Advantages

- Reduced coupling case: changes in the submodule A inside C impacts only Module C.
- High-coupling case: if one changes A, then one must consider the effect on submodule B.
- So, high coupling complicates system design because:
  - high degree coupling has cascading effect where a simple change cannot be made in isolation.
  - coupling forces the designer to consider the whole module and understand everything in order to ensure that a change does not break something



Reducing coupling : Disadvantages

1. **Design area increased**
   - We traded an explicit measure of quality (design size) for a subjective improvement in another measure of quality (maintainability).
   - However, this change does not necessarily increase the design size!
2. **The designer now has to maintain the same component in two places.**
   - So if a bug is discovered in submodule A, then it is fixed once in the first design.
   - However, in the alternative design, one has to fix the bug in both submodules B and C.



3.1.5. Designing for Reuse

- In addition to improving design quality we want reusable components.
- Construction designs with the foresight that components or even the whole design will be reused again in the future
- what is necessary to create reusable designs?
- One indicator is **high cohesion** and **low coupling** which leads to reusable design components
- must also consider costs in terms of:
  - RCR — relative cost of reuse
  - RCWR — relative cost of writing for reuse

RCR vs. RCWR

- RCR — you have to read and understand how to use the class hierarchy before you can reuse its code
- RCWR — someone has to put a lot of effort into designing a hierarchy for reuse

For Example:

- writing a C program to copy character data, say 32-Bytes worth of data, we could write our own for-loop to exactly copy the 32-Bytes of data
- could reuse this loop and augment it to support various sizes; however, we are limited to copying characters, or Bytes
- alternatively, there is already a more efficient libraries in C, such as string where strcpy already exists
- trade-off is learning how to use strcpy compared to the time it takes to create your own copying function is RCR

RCR and RCWR in System Design

- in some cases, it could be easier to generate your own in place learning a potentially complex component
- this leads into RCWR (the cost associated with making your custom create component fully reusable)
- one way to manage RCWR is with an incremental approach:
  - design a specific component for the current design
  - if it is needed again, copy-and-generalize it
- in VHDL this could be as simple as adding generics to the design

Refactoring

Refactoring is the task of looking at an existing design rearranging the groupings and hierarchy without changing its functionality

- Example: see coupling reduction example.

Testing

- test is very IMPORTANT!
- the value of reusable components is clear
- the danger is a refactore component accidentally changes functionality
- **regression testing** is automated and might be simulation driven (a la test benches) or it may be a set of systems that wraps around the component and exercises its functionality

Outline

3.2 Control Flow Graph

- Software Reference
  - We will discuss the advantages of software reference
  - Recall ISS in systems programming class.
- In this chapter, the Control Flow Graph (CFG) is used to visualize the software reference design.
- So, advantages of CFG:
  - Graph algorithms applied to the CFG will result in a set of properties that **guide transformations (optimizations) designed to improve the performance** of the program.
  - When combined with data dependence (see Chapter 5), the CFG can be used to determine what parts of the program can be executed **in parallel (and thus implemented spatially in hardware)**.

Software Reference Design

- ***software reference design*** is a rapid prototype — a piece of software that functionally mimics the behavior of the whole system, even hardware modules that have not been implemented yet.
  - **Disadvantage**: the cost associated with creating it
  - **Advantages**:
    - It is generally a complete specification.
    - It is executable— a designer can gather valuable performance data from running the software reference design with specific input data sets.
    - Because the software reference design is computer readable, the specification can be analyzed by existing software tools.

Software Reference Design

- Through the next several chapters we will assume that a software reference design exists.
- The computation in software reference design can be represented mathematically.
- This would help make decisions about what parts of the system should be implemented in hardware versus software.
- We do this by borrowing some concepts from compiler technology— primarily the control flow graph

Control Flow Graph: assembly

- Control Flow Graph (CFG) is graph G = (V, E)
- The vertices (or nodes) V of the graph are basic blocks and the directed edges indicate all possible paths that program could take at run time.
- A basic block is a maximal sequence of sequential instructions with a Single Entry and Single Exit (SESE) point.
- In the figure:
  - The first group of instructions (A) is not a basic block because it is not maximal — the first instruction (store word with update) should be included.
  - The second group of instructions (B) is a basic block.
  - The last group (C) is not a basic block because there are two places to enter the block (at the store word instruction after the add immediate, or by branching to label .L2).

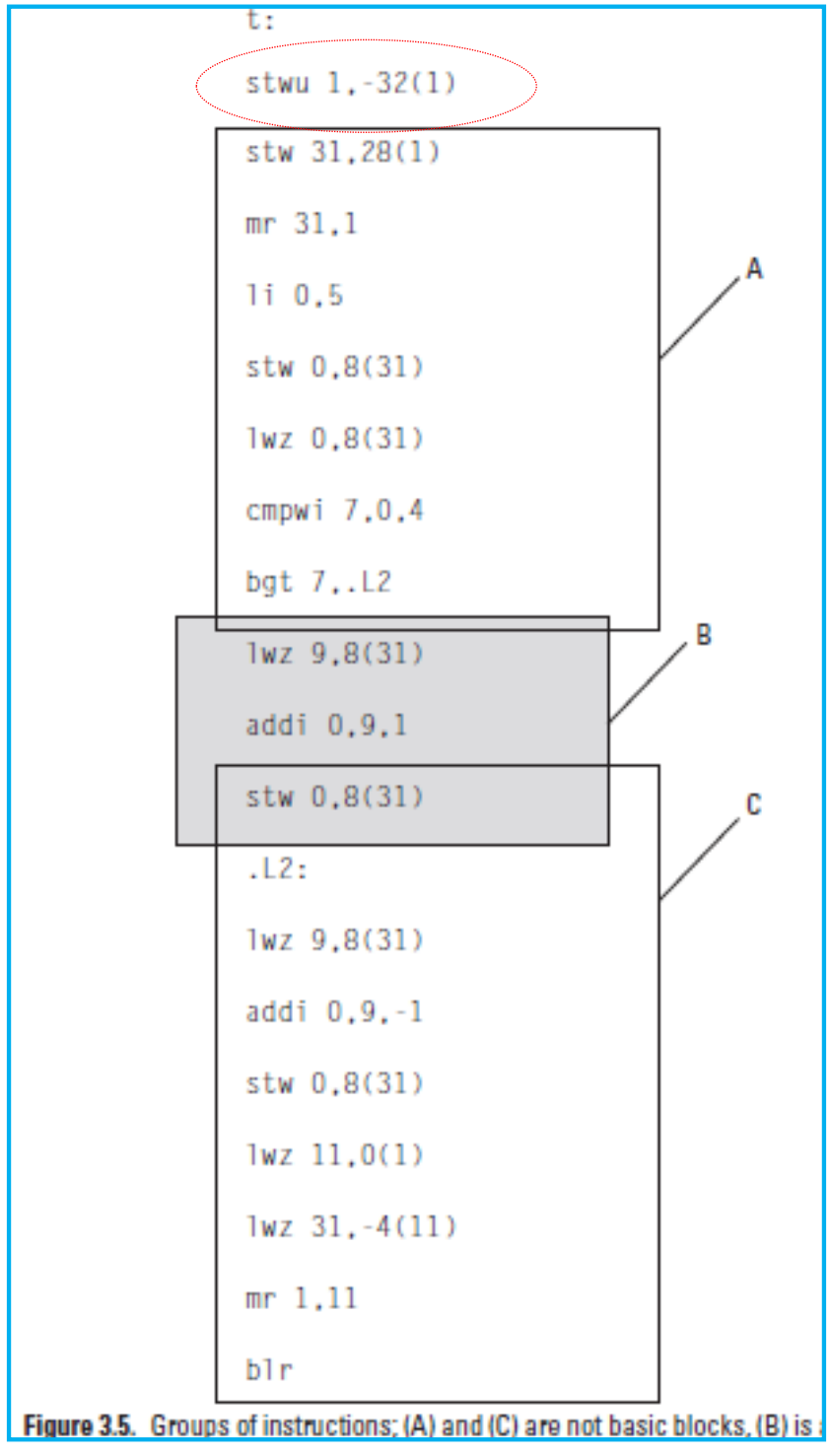
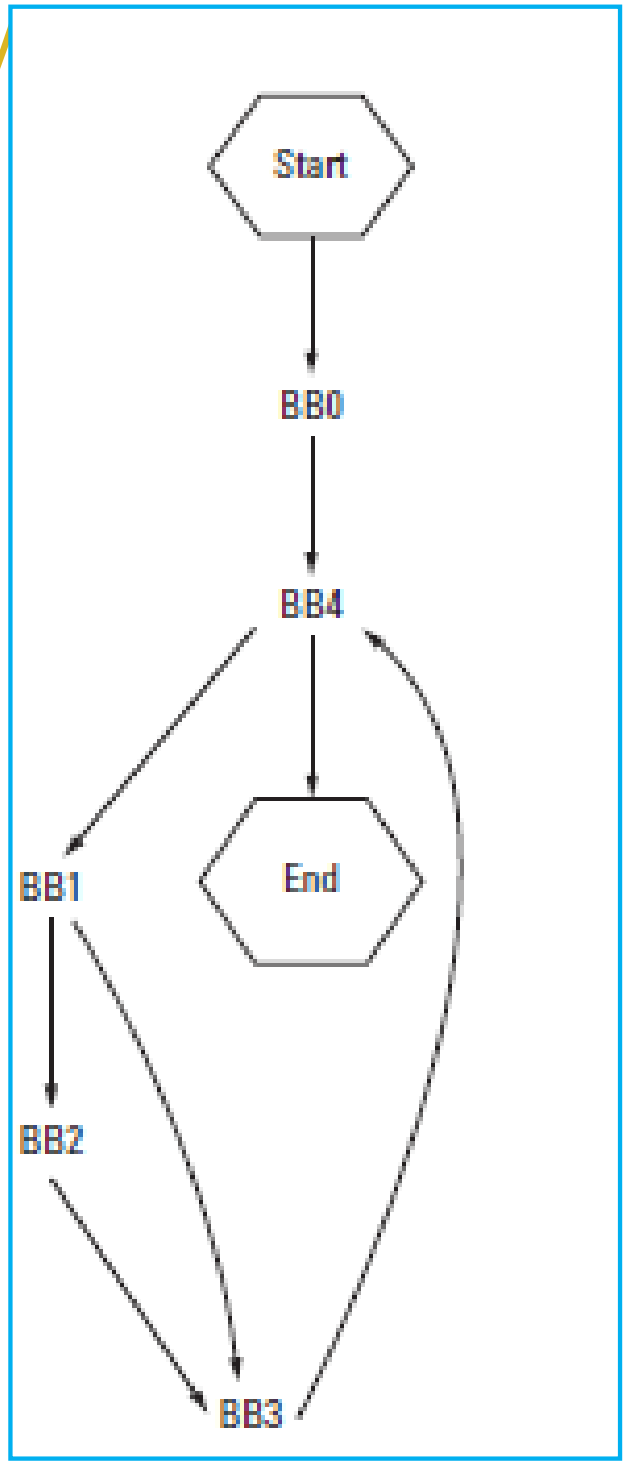


Figure 3.5. Groups of instructions; (A) and (C) are not basic blocks, (B) is

Control Flow Graph for assembly code

- An edge *(b1, b2) in a control flow graph indicates that after executing* basic block *b1 the processor may immediately proceed to* execute basic block *b2.*
- *If the basic block ends with a conditional* branch, there are two edges leaving the basic block.
- If it does not end with a branch or if the last instruction is an unconditional branch, there will be a single edge out.
- Two special vertices in the graph, called *Entry and Exit, are always added to the set of basic* blocks. They designate the initial starting and stopping points, respectively.

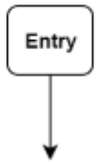


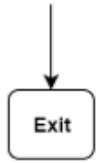
CFG from application program

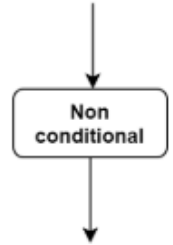
- Through the next several chapters we will assume that a software reference design exists.
- The computation in software reference design can be represented mathematically.
- This would help make decisions about what parts of the system should be implemented in hardware versus software.
- We do this by borrowing some concepts from compiler technology— primarily the control flow graph

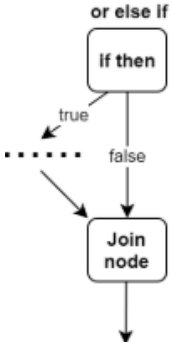
Flow Graph: Program code

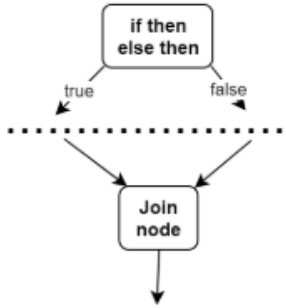
- Program code is typically pseudo-code i.e. close to C-code
- Linking code to vertices:
    - In some cases, codes typically writing in the vertices.
    - Others, code is marked with vertices names.
- Below is rough guideline of translating loops/if to graph.
- Add the special vertices for *Start* (*Entry) and End (Exit)*

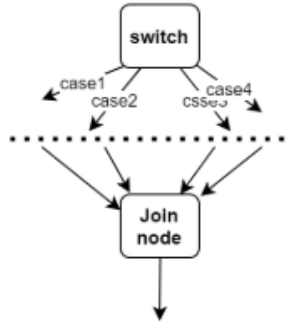


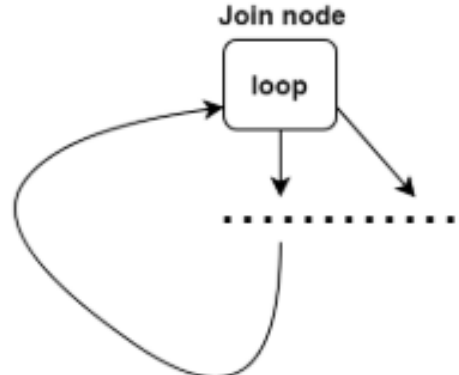












Control Flow Graph: Example (C code)

- Figure (a) shows simple subroutine in C.
- Figure (b) identifies the basic blocks in powerPC assembly code.
- Figure (c) shows the control graph.

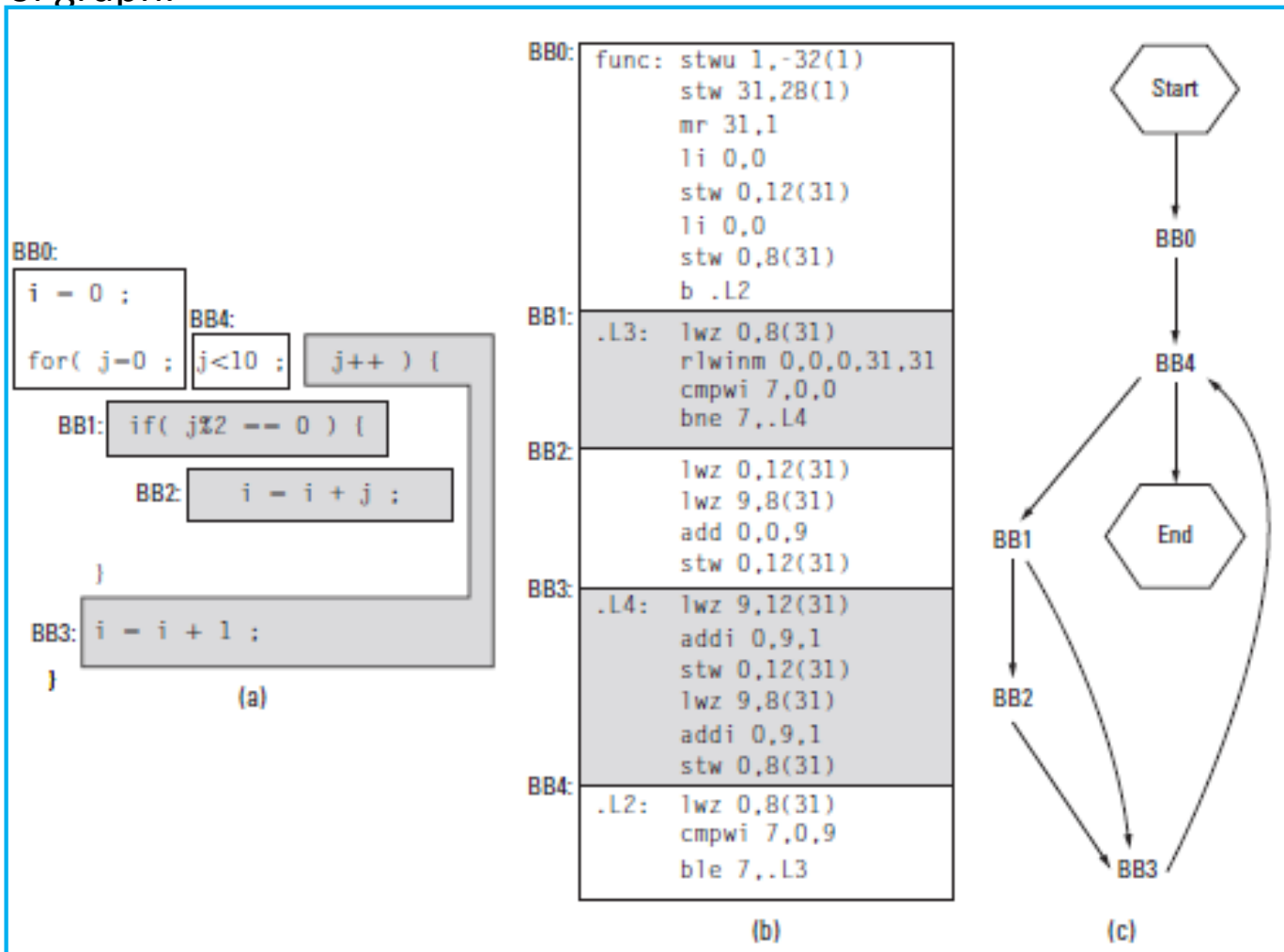
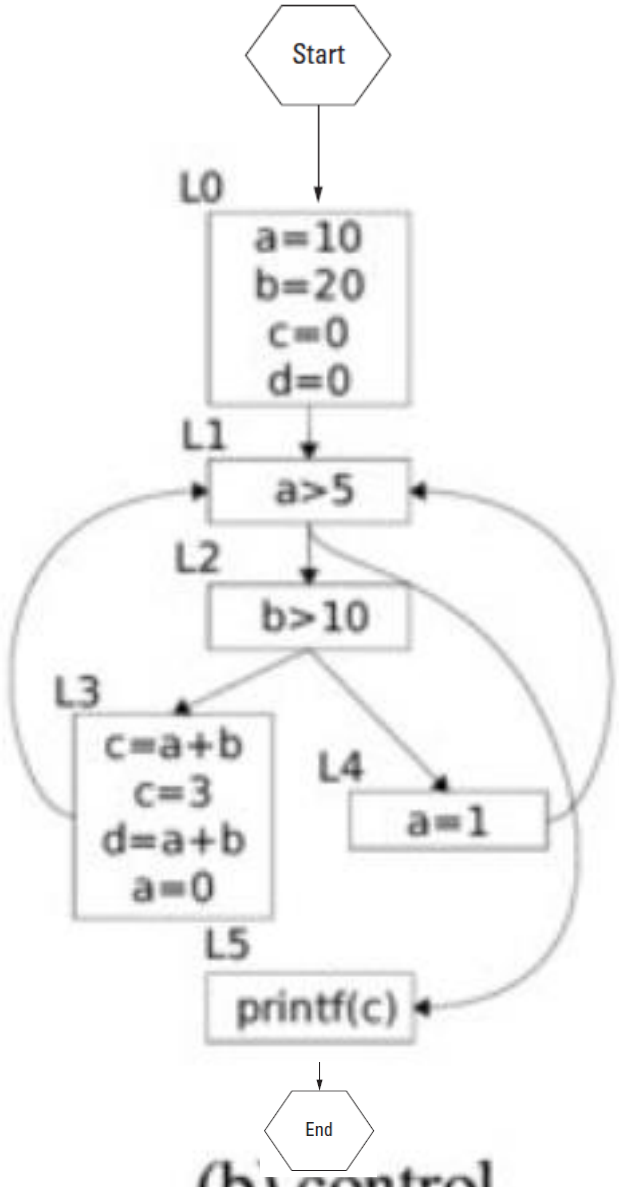


Figure 3.6. Basic blocks in (a) C source code (b) translated to assembly (c) a control flow graph.

Control Flow Graph: Code Example

```
a=10;
b=20
c=0;
d=0;
L1:
if(a>5){
  if(b>10){
    c=a+b;
    c=3;
    d=a+b;
    a=0;
    goto L1;
  }
  else{
    a=1;
    goto L1;
  }
}
else{
  printf("%d\n",c);
}
```

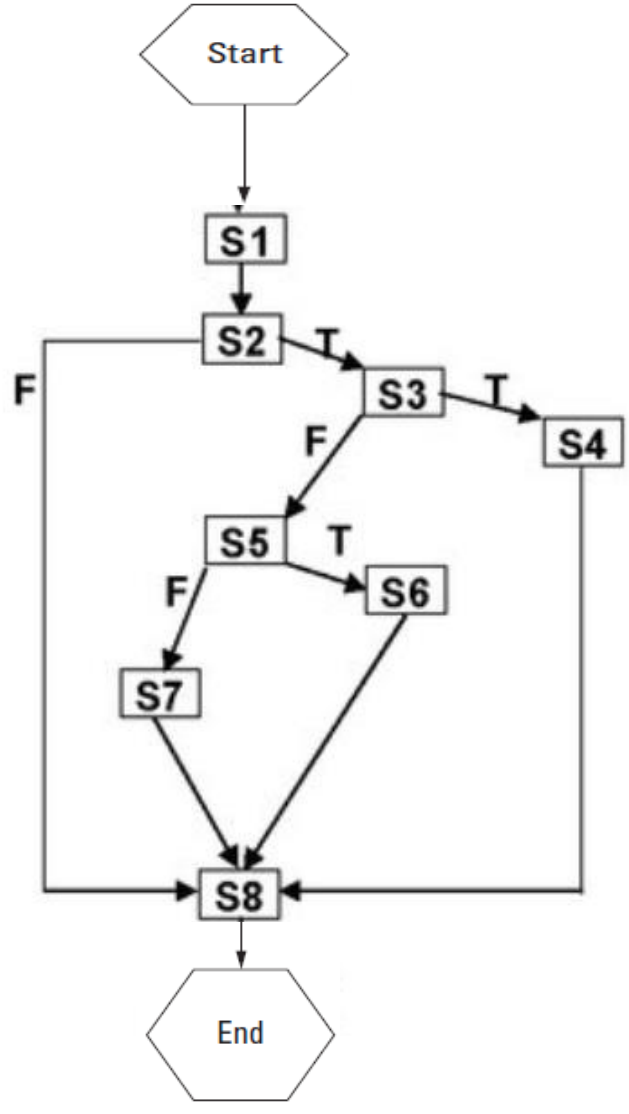
(a) source program



(b) control flow graph

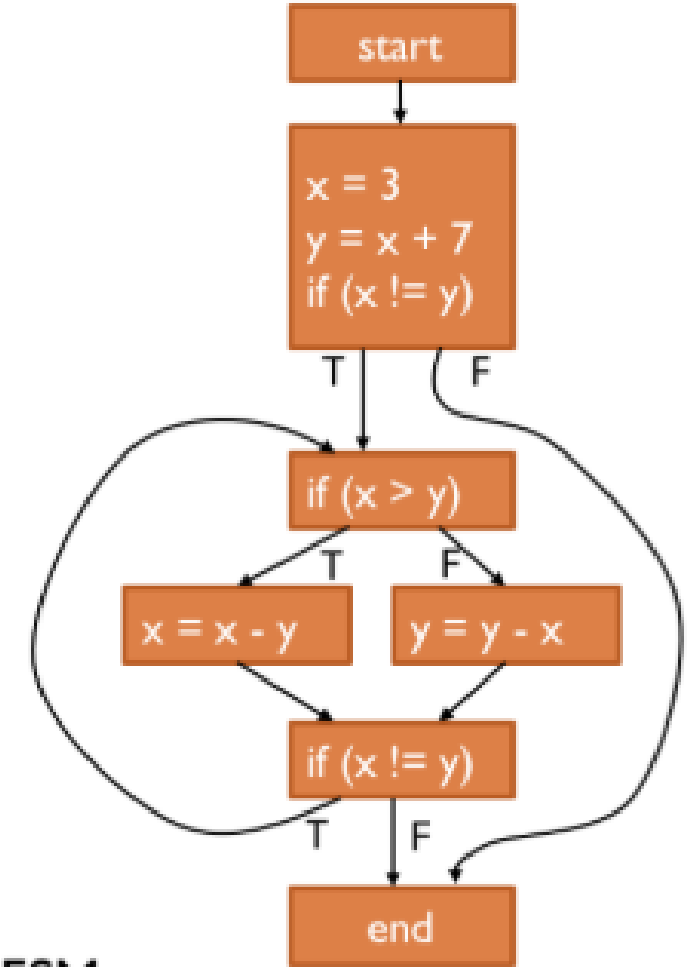
Control Flow Graph: Code Example

```
Procedure TriangleType
 S1   char *returnString = "not triangle\n";
 S2   if (((i + j)> k)&&((j + k)> i)&&((k + i)> j))
 S3      if ((i + j + k) == (3 * i))
 S4         returnString =  "equilateral\n";
         else
 S5         if ((i != j) && (j != k) && (k != i))
 S6            returnString = "scalene\n";
            else
 S7            returnString = "isosceles\n";
            endif
         endif
      endif
 S8   return returnString;
```



Control Flow Graph: Code Example

```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
    x = x - y;
  } else {
    y = y - x;
  }
}
```



Difference between Control flow Graph Data flow Graph

	Control Flow Graph	Data Flow Graph
Representation of	Program/process	Data dependency
Usage	Illustrates program function/flow	Illustrates input/output data and how data is passed between computations.

Graph components	• Nodes:
	  ○ Start/end hexagons (or rectangles)
	  ○ rectangle used to represent a sequential computation (multiple operations),
	  ○ a decision box labelled with T and F to represent True and False evaluations respectively
	  ○ a merge point.
	• Arcs: represent control flow
		• Nodes: denote operators
		• Values: input/output/intermediate values.

Main differences	• starts with Start/Enter node and ends with End/Exit node • sequential rectangles can include more than one operations • decision box represents conditions	• start with input and ends with outputs • each node single operation • no conditions

Outline

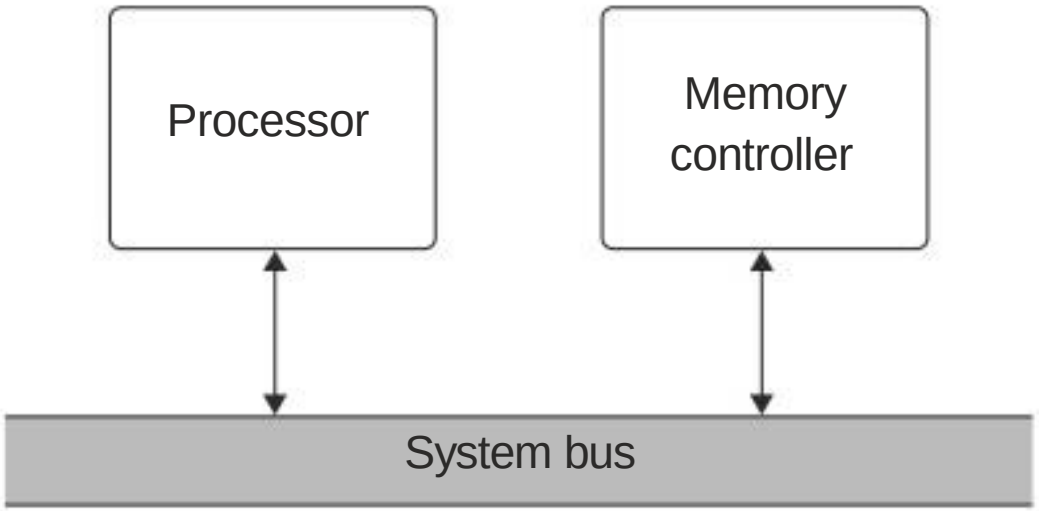
3.3. Hardware Design

- Now, we turn our attention towards hardware design
- specifically hardware components available to Platform FPGA designer
- a brief description on the evolution of these components
- followed by a description of:
  - processors
  - memory
  - buses and bridges
- conclude with a description of how to use custom hardware modules

3.3.1. Origins of Platform FPGA Designs

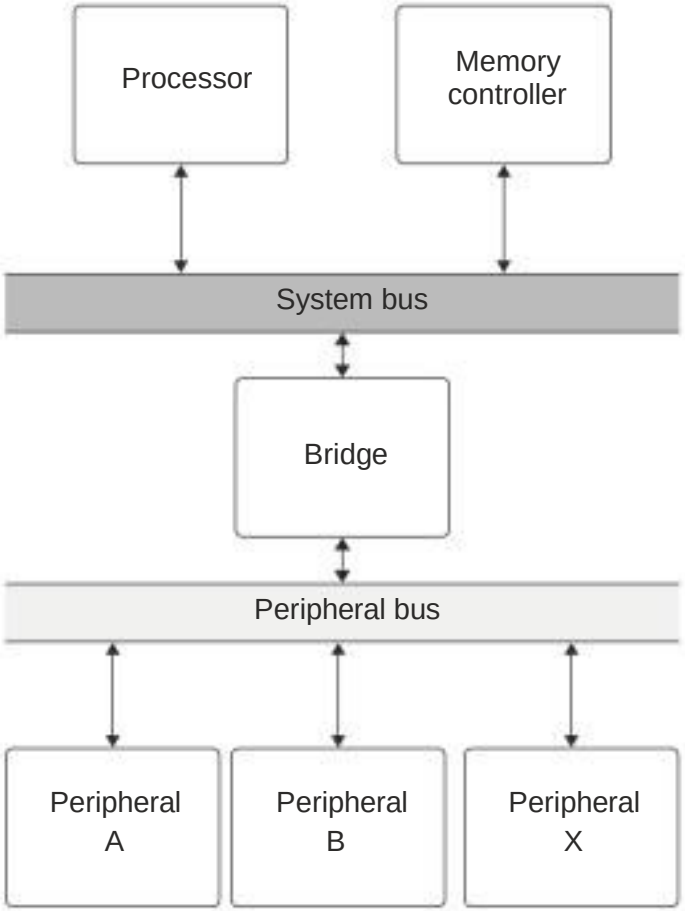
- designers **rarely** want to build embedded systems from **scratch**
- to be productive designer will typically **begin with an existing** architecture
- then remove unneeded components
- and add cores to meet the project requirements
- one model that is commonly used as a base design is the processor-memory model

Processor-Memory Model



the fundamental process-memory model that is commonly used as a base system in Platform FPGA designs

Two Bus System



the two bus process-memory model used to support parallel, independent high speed and low speed communication

Processor-Memory Model on FPGAs

- Organization is a good starting point (or platform) for FPGA designs
- Can utilize hard or soft processor cores,
- On-chip and off-chip memory controllers,
- and peripherals (existing) cores to interact with the system
- quickly can amass a system that resembles modern computer architecture

Why Processor-Memory Model on FPGAs?

Platform FPGAs have adopted these two basic processor memory model from the desktop computer architecture because it:

- provides an established framework custom designs can be built on top of
- shortened development time using existing infrastructure
- the designer can focus on new/custom cores
-

Why Processor-Memory Model on FPGAs?

"Why create a generic base system when we are using FPGAs?"

- in an ideal world, could create custom design from scratch
- in reality projects have deadlines and budgets.
- Can leverage existing software infrastructure (such as Linux)
- FPGAs being field programmable allow a less than ideal design to be released first, then updated later with a more ideal design

3.3.2. Platform FPGA Components

- in Chapter 2 we discussed the FPGAs basic building blocks:
  - function generators
  - logic cells
  - logic blocks
  - on-chip memory
- use these components to assemble larger systems
- with the ideas of modularity, cohesion and coupling of components and designs, want to **build base systems.**

Base systems are used and reused as a starting point for embedded systems design.

Processor

- is an obvious **starting point** when describing the processor-memory model
- offers the designer control and a familiar design environment
- useful in **rapid prototyping**
- as more of the design is offloaded to the FPGA the processor's role may reduce
- two types of processors may exist, **hard and soft**
- Chapter 2 gives more details on the differences between the two

Soft Processor

- if no hard processors exist on the FPGA, a soft processor must be used
- - this **requires** sufficient reconfigurable **resources**
- + soft processors offer a **great** deal of **flexibility**
- can be configured to more specifically meet the needs of the design
- i.e, if a Memory Management Unit (MMU) is needed, one can be designed/added
- soft processors also allow for **incremental improvements**
- i.e, the Xilinx MicroBlaze has seen significant improvements

Processor's Capabilities

- processors can operate in standalone mode offering most basic functionality (i.e, stdin/out)
- with a MMU processor can support more verbose Operating Systems
- multi-processor support with coherency and shared memory
- however, knowing the processors role is key to the design
- i.e, PicoBlaze is great for complex state machines, but cannot support Linux

Choosing a Processor: soft or hard core

Before choosing a processor, consider the following questions:

- does the FPGA include hard processors?
- are there sufficient resources to implement a soft processor? **No, use hardcore processor.**
- what role will the processor play in your design?
- what type of software will be used on the processor?
- how much time will the processor be used vs. hard cores?
- are there future plans to using a different FPGA?

Yes, use soft-core processor.

Memory

- in order for the processor to do any useful work, memory must be included to store operations and data
- memory hierarchies and organization is important to consider
- Von Neumann **(one memory for instruction and data)**
- Harvard architecture **(two separate memories for data and inst.)**
- cache (L1, L2, etc.)

Choosing Memory

Before choosing memory, consider the following questions:

- what type of memory is available?
- is there on-chip and/or off-chip memory?
- how much on-chip/off-chip memory is available?
- is the memory volatile or non-volatile?
- how does the processor interface with the memory?
- how does the system interface with memory?
- how do specific cores interface with memory?
- is the memory being utilized efficiently?

Choosing Memory: on-chip vs. off-chipe

- modern FPGAs include varying amounts of on-chip memory
- can also use flip-flops for sort term storage (in small quantities)
- can include memory in a core
- as part of the base system connected via a bus
- or off-chip connected through a more complex memory controller

Choosing Memory

- how memory is interfaced is important
- low latency, high bandwidth, and/or storage capacity
- efficient memory controllers are necessary
- consider how memory is connected to the compute core
- i.e, a 400 MB/s bus is not ideal for a 6.4 GB/s memory
- reuse of memory controllers is key — design once and reuse!
- Chapter 6 covers memory in much more detail

Buses

- The processor(s) and memory controller(s) connect to the bus via a **standard bus interface**.
- The bus **signals** mainly consists of:
  - Address,
  - data,
  - read/write requests,
  - acknowledgment signals.
- The bus can **include**:
  - bus arbiter, which controls access to the bus.
  - Masters: A master is a core that can request access to the bus
  - Slaves: a slave responds to bus transactions
- **Two-bus system**: used to isolate lower speed peripheral devices from higher speed devices.

Choosing a Bus

Before choosing a bus, consider the following questions:

- what cores will need to directly communicate?
- do certain cores communicate more often than others?
- do specific cores require higher bandwidth between them?
- can any cores function on a slower bus?

Bus Types: System Bus and Peripheral Bus

- System Bus
  - **Highest bandwidth** bus that **connects the processor, memory** controller and high speed devices.
- **Number of cores** and Bus Type:
  - When the number of cores needing access to the bus is relatively **small**, connecting all of the cores on a single bus is a logical, resource efficient decision.
  - As the **number of hardware cores grows**, a decision must be made as how to most efficiently support these additional cores in combination with the already existing system. One solution is to introduce a second bus: peripheral bus.

Peripheral Bus

- Peripheral Bus is a second bus added to separate the design into different domains. For reasons such as:
  - Separate high-speed and low-speed designs.
  - Provide a subset of the cores with a dedicated bandwidth for communication.
  - Cores in the second bus can communicate in parallel (with activities in the system bus.)

Bridge

- A bridge is:
  - a special core that **resides on both buses** and propagates requests from one bus to another.
  - It functions by interfacing as **a bus master** on one bus and a **bus slave** on the other.
- Bridge types:
  - A system-to-peripheral bridge is used when the peripheral bus only will respond to requests from the system bus.
  - A peripheral-to-system bridge is added if cores on the peripheral bus need access to the system bus (say for access to the off-chip memory controller).

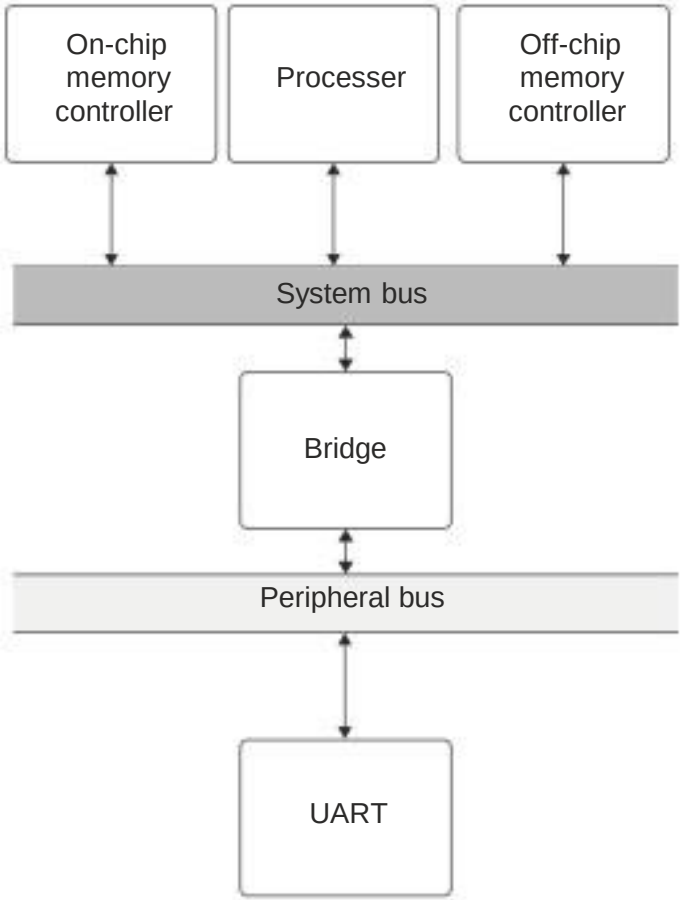
Peripheral

- Peripheral cores are hardware cores that interface to peripheral devices, such as printers, GPS devices, and LCD displays.
- Examples of peripheral cores:
  - PCI bridge and PCI arbiter
  - Ethernet core
  - USB core
  - UART
  - I^2C and SPI

Building a Base System

- **Components:**
  - We want to build a simple base system, consisting of a process, on-chip memory, off-chip memory, and a UART for serial communication with a host PC.
  - So, we have a processor, two types of memory, and a UART.
- **Buses:**
  - We can connect these components together because that is a little more application specific (i.e., not flexible).
  - However, we want to be **flexible enough in our design to allow custom cores to be added without requiring significant changes to this base system**.
  - For that fact, we will include **both a system bus and a peripheral bus.**
- **Bridge:**
  - we need to include a bridge: peripheral-to-system bridge

Base System



block diagram of the base system consisting of a processor, on-chip and off-chip memory, and a UART

Building the Base System

- impractical to use **schematic capture** software for this design
- use hardware description languages
- use bottom-up design approach
- end result, one large HDL file with each component instantiated within it
- must also include pin constraints for UART, clocks, off-chip memory
- use existing components when available
- most vendors provide tools to simplify these steps
- Xilinx offers a Base System Builder (used in Chapter 1)

Adding to Platform FPGA Systems

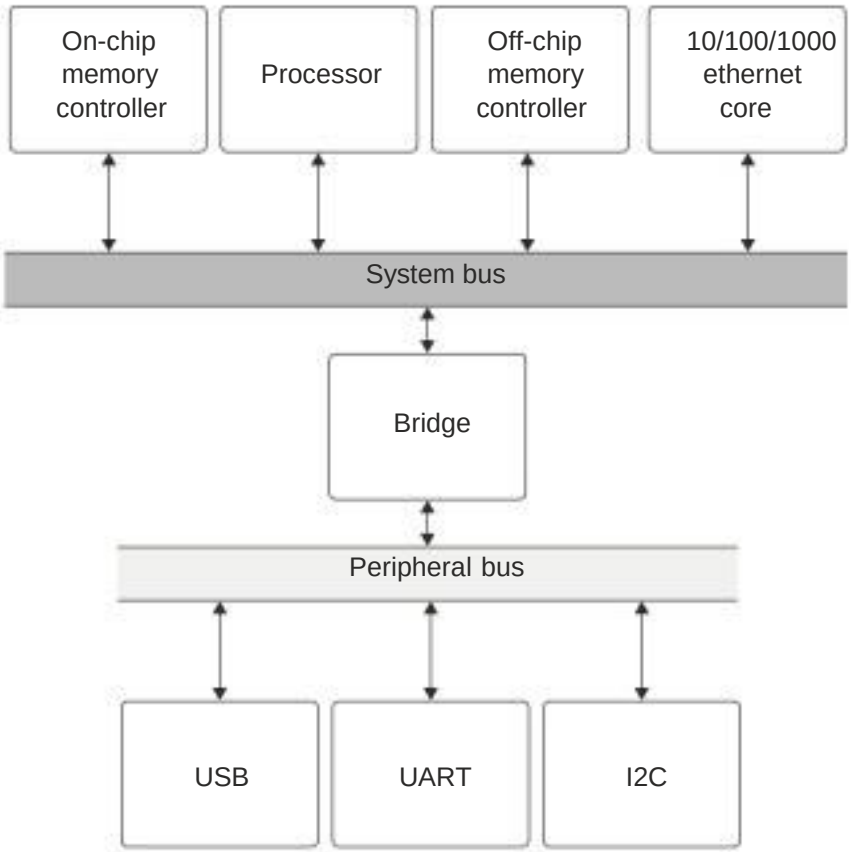
- a base system is nice, but let's add to it
- there are many cores, which one to add next?
- many designs need **network (Ethernet)** access
- the network core will provide access to the FPGA via a web interface
- other common cores may be **USB** to generic peripheral attachment
- and **I2C** or SPI for low-speed peripherals on the same PCB

Adding to Platform FPGA Systems

Adding another core typically includes:

- instantiating the core in the design
- connecting the core to a bus (or other cores)
- adding custom pin constraints for cores with I/O
- Chapter 7 goes into more details regarding FPGA designs with I/O

Modified Base System



block diagram of the base system with the additional cores: networking, USB and I2C

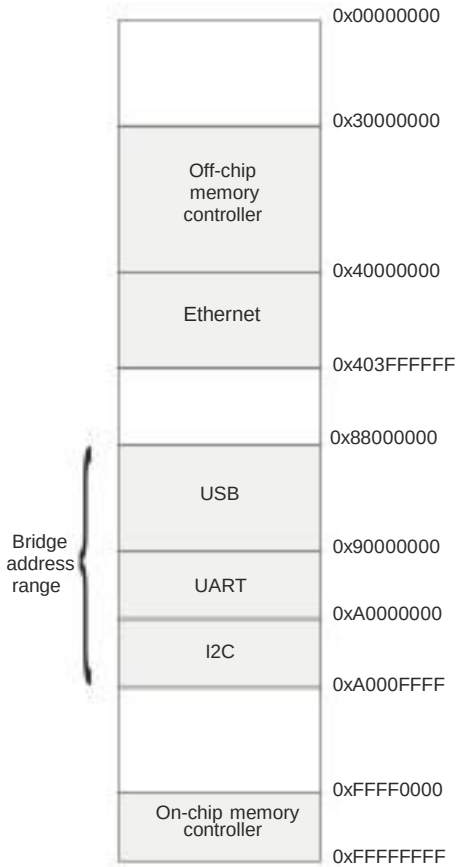
Address Space

- physically speaking the “location” of data is straight forward and easily identifiable
- off-chip memory is located outside of the FPGA’s packaging
- on-chip memory is stored in SRAM cells within the FPGA’s fabric
- from a design perspective accessing the memory is done through an address space

Address Space

- Off-chip memory may be addressable from the address range 0x30000000-0x3FFFFFFF, for a total of 256 MB of addressable data
- globally addressable on-chip memory, located on the same bus as the off-chip memory controller, may have an address range 0xFFFF0000-0xFFFFFFFF, for a total of 64 KB of addressable data
- in a platform FPGA design it is possible to **automatically generate these address ranges or to set specific address ranges**

Address Map



one possible address map for the theoretical two bus system with networking, USB, I2C and UART

3.3.4. Assembling Custom Compute Cores

- “why build custom compute cores?”
- advantages?
  - speed
  - efficiency
  - predictability
- disadvantages?
  - cost
  - design time / constraints
  - achievable performance

Why Build Custom Cores?

- hardware development is hard
- fewer hardware engineers than software engineers
- processors are inexpensive/cost-effective at scale
- common rebuttal: “performance” (or “speed”)
- we will also consider efficiency and predictability
- important for a hardware engineer to know when it is and is not justifiable to implement a custom core

Advantage: Speed

- The following are the practical reasons as why FPGAs have been able to outperform standard processors.
- First: the **execution model**
  - The sequential computing model of the standard von Neumann processor creates an artificial barrier to performance by expressing the task as a set of serial operations.
  - With hardware, the task's inherent parallelism can be expressed directly.
  - Also, hardware does not need to fetch instructions, they are part of the design.
- Second: **Specialization**
  - In general purpose processors, data paths and operation sizes are organized around general requirements.
  - FPGA-based implementation can be created with customized function units to meet the exact needs of the application.
- Third: FPGA design facilitates **run-time reconfiguration.**

Advantage: Efficiency

- Suppose a hardware implementation of task A takes exactly the same amount of time to complete as the equivalent task executed on a processor. We can assume the software implementation is easier to develop. Is there any reason to build a hardware implementation?
  - Yes, when the hardware is solution is more efficient
- By efficiency, what we are concerned with **is how to accomplish a fixed task with a variable amount of resources**. e.g. area on a chip, the number of discrete chips, or the cost of the solution.
- A hardware design plus a processor is **often more efficient** than two processors.
  - If both approaches meet the minimum criteria, then we say the more efficient solution is the one with a lower cost.
  - If the system is being deployed on a single chip, then the more efficient solution is the one that uses less area.

Advantage: Predictability

- Timing constraints are very important.
- When there are **real-time** demands on the system, scheduling becomes important.
- **Hardware designs have the benefit of being very predictable**, which in turn makes scheduling easier.
  - custom hardware cores can be designed with predictability in mind
- For real-time systems, where the goal is to satisfy all of the constraints, predictability is often more valuable than simply making the system faster.

Disadvantages

- First: the **development effort** required
  - Unless there is a compelling reason (in terms of the performance metrics from Chapter 1 or the advantages just mentioned), then it may not be worth the extra effort.
- Second: **the loss of generality**
  - It is simply the nature of our modern world that product requirements will evolve over time. The loss of generality has the potential of negatively impacting the design over time.

Design Composition

- Two ways of building digital computing machines.
  - First: “build from gates”
  - Second: modular design approach
- The first begins with logic gates (i.e., build from gates).
  - Begins with requirements that are translated into a specification (expressed in various forms such as Boolean expressions, truth tables, and finite state machines).
  - Then use formal techniques to reduce the number of states, minimize the Boolean functions, and realize the machine in the fewest number of components.

Design Composition, cont.

- The second starts with higher level logic blocks from which complex systems are composed.
  - The designer starts with logic blocks that have a predetermined functionality — such as decoders, multiplexers, and flip-flops.
  - These components are selected and arranged creatively to meet the requirements.
  - The second approach is what we will focus on for the remainder of this course.

Modular Design Steps

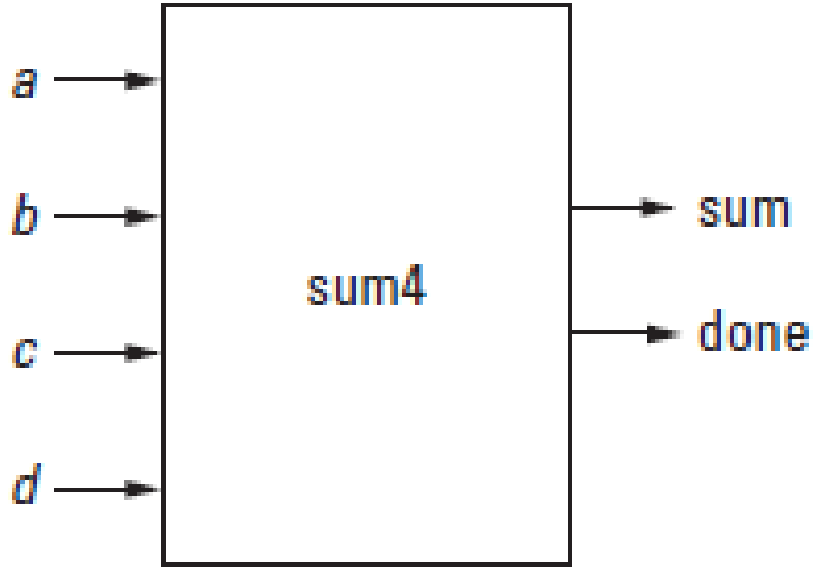
1. The first step is to identify the **inputs and outputs** of the core.
2. The second step is to identify the operations and compose **a data path**.
   - The data path represents the flow of data through the component.
   - Once the flow has been established it becomes possible to construct a computation pipeline.
3. The third step is to develop a **controlling circuit** that sequences the operations, usually a finite state machine.

Bottom-Up and Top-Down Design

- **Bottom-up approach** is used when assembling systems.
  - This same approach can be used for **assembling** custom compute cores.
  - Each subcomponent can be treated as a black box, where only inputs and outputs are known to the designer.
- **Top-Down**: starts from the top-level design and working down to low-level components
  - The designer would begin with the core's interface (inputs and outputs).
  - This creates a black box representation of the core.
  - Once the interface is set, the designer can begin to systematically **decompose** the design into its subcomponents.
  - This process is repeated for each subcomponent until low-level components are identified to be simple enough to design.
- Next, will consider an example to **illustrate the different ways components can be combined to form large modules**.

Design Example

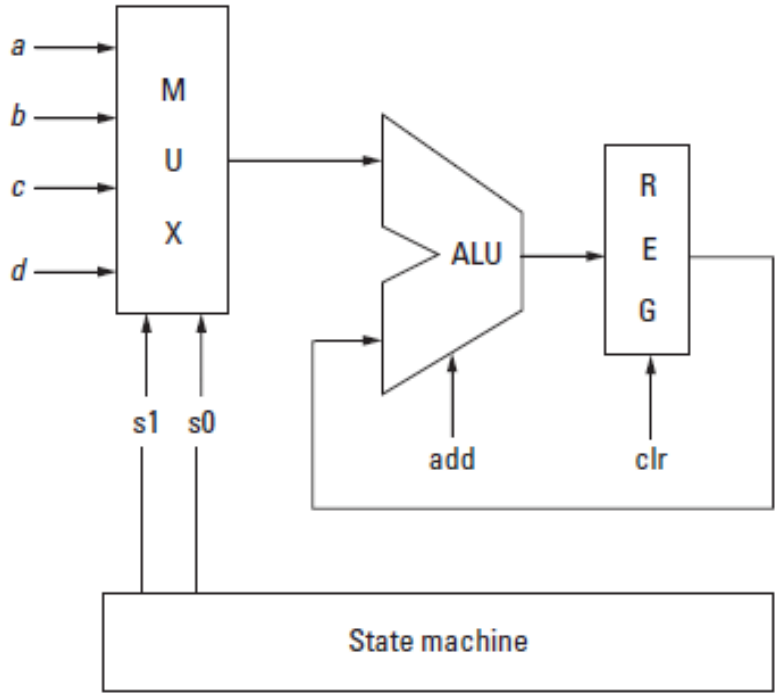
- The desired functionality is to add four numbers together.



. The top-level component to add four numbers.

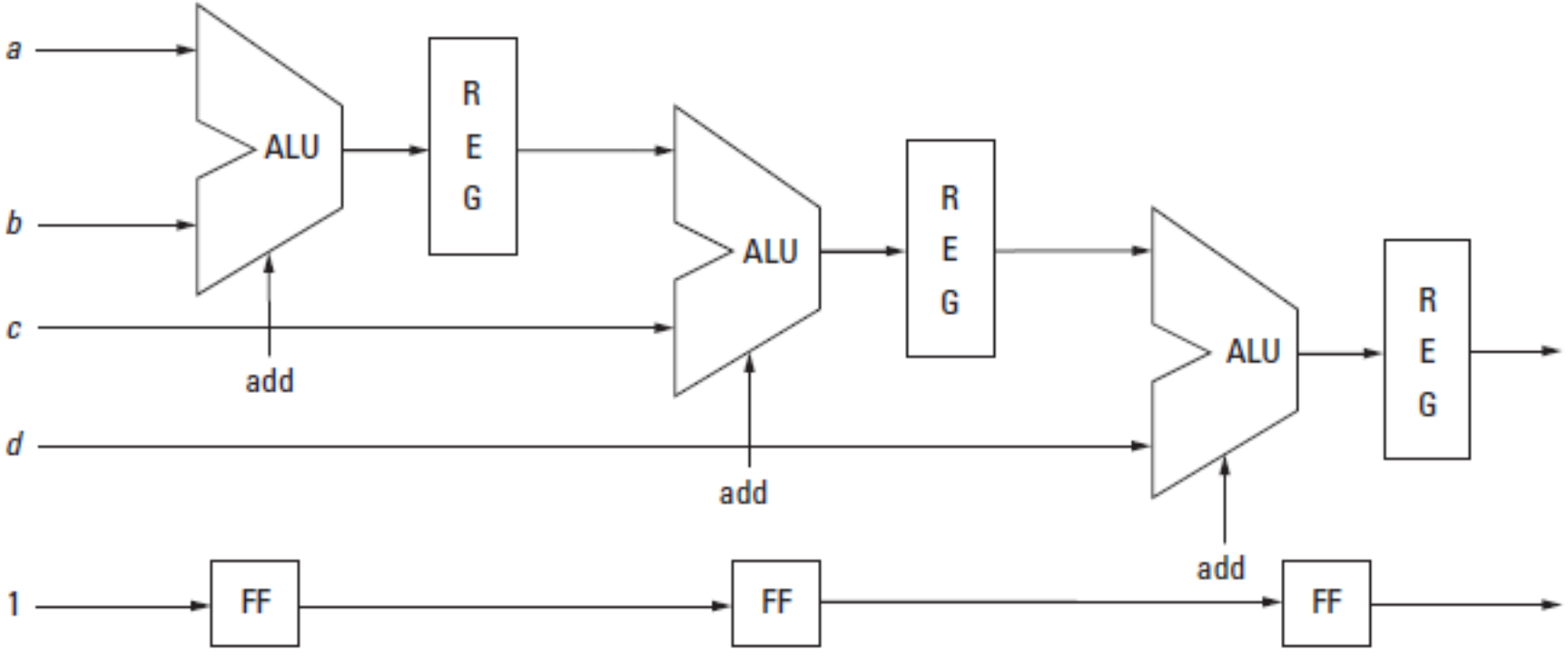
Temporal Implementation

- The addition would take four additions:
  - 1. regt0 = 0+a
  - 2. regt1 = regt0 +b
  - 3. regt2 = regt1 +c
  - 4. regt3 = regt2 +d
- In a system were only one ALU and register exist, this would be **a sufficient minimal resource solution**.
- Clearly this is **not the fastest** approach. In terms of speed it is desirable to perform as many independent operations in parallel as possible.
- Unfortunately, there is a cost with parallel approaches, that is, added resources.
- Using three ALUs we could perform temp1 = a + band temp2 = b + c in parallel and then add temp1 + temp2.



Spatial Implementation

- In this case, the additions are **pipelined** such that results are fed forward to the next adder.
- **Concurrency** is what gives system designers speed, and control of timing is what gives system designers predictability—both primary motivations for using hardware.



Outline

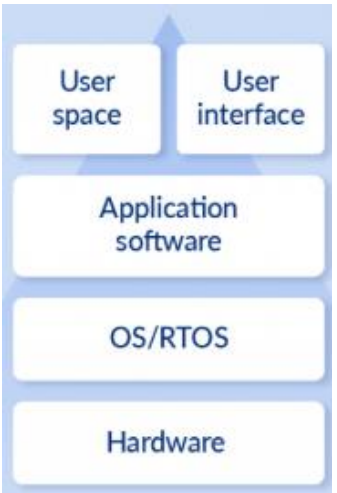
3.4. Software Design

- embedded systems software rapidly becoming more **sophisticated**:
  - •includes **memory management unit**, virtual memory
  - •can run **full operating systems** on devices
  - can adopt applications originally designed for general purpose desktops
- this can be good or bad - caution needed!
- section aims to cover background information for software implemented on Platform FPGAs

3.4.1. System Software Options

- **system software** is any software that **assists** the application — usually by adding a software interface to access the hardware
- ranges from simple library of routines
- to full-fledged Operating Systems that virtualizes the hardware for individual processes

System software:
- **OS**
- **Drivers**



Embedded System

System Software Options: “no system software”

- **C Start-up files:**
  - the C start-up files (subroutines that the compiler/linker adds to every C program) are modified.
  - At run time, these subroutines execute before calling the main function of the designer’s application.
  - With no operating system, these initial subroutines are responsible for **setting up the processor and peripherals**.
- **Standalone C** program runs without the support of any additional system software. The advantage of this approach is that there is essentially no overhead.
- **Advantages:**
  - The solution produces a small enough executable that the entire software
  - system can fit within the On-chip memory . Avoiding off-chip RAM can be a significant advantage for some embedded systems.
- **Disadvantages:**
  - Does not support software development developer
  - Difficult to take advantage of existing software that assumes a full C library

System Software Options: “additional functionality”

- Numerous products and Open Source solutions are available that specifically target embedded systems to meet this need.
- They range:
  - from simply **adding timer interrupt service routine**
  - and the ability to switch between different **threads** of control
  - to full-featured **operating systems** that only lack a memory management unit.
  - In some cases, **the system software is combined with the application** when the application is compiled

Root Filesystem

UNIX and its variants (Linux, BSD, Solaris, and many, many others) share the concept of a root filesystem

- a **filesystem** is a data structure implemented with a collection of equal-sized memory blocks that provides the application with the capability of creating, reading, and writing variable-sized files
- most filesystems organize files hierarchically
- in UNIX, files and sub-directories are grouped in directories
- one special directory called root which contains files and sub-directories
- the filesystem data structure is most often implemented on secondary, non-volatile storage

Root Filesystem

- when a file system is being manipulated it is typically referred to as a file system image
- when a filesystem is being used to manipulate files it is called a mounted filesystem
- in some cases the designer must create some initial filesystem called the root filesystem
- during boot the kernel interacts with the filesystem
- once running the kernel calls a special application called init which is the first process to run
- the init process finishes booting the system with the configuration files stored in the root file system
- what this means is that the embedded system designer has to know how to create a filesystem and know how to populate it
- Lab 3 goes into specific details with step-by-step directions

Linux on a Platform FPGA

We refer you to Lab 3 and the Gray Pages for specific details

Chapter-In-Review

- principles of system design and the hardware and software background that is necessary to be able to construct embedded system designs on a Platform FPGA running with a full-fledged operating system
- emphasized important design concepts to support the ability for base systems, custom hardware cores, and low-level components to be reused within a system
- discussion of system software and its role in Platform FPGA design

Chapter 3 Terms

external criteria are characteristics of a design that an end-user can observe

internal criteria are characteristics that are inherent in the structure or organization of the design, but not necessarily directly observable by the user

correctness refers to a system that has been (mathematically) shown to meet a formal specification

reliability in terms of hardware means that the system behaves correctly in presence of physical failures

resilience accepts the fact that there will be errors and the design “works around” problems even if it means working in a degraded fashion

Chapter 3 Terms

verifiability is the degree to which parts of the system can be formally verified, i.e., prove correct

maintainability refers to the ability to fix problems that arose from unspecified behavior

repairability refers to fixing situations where the behavior was specified but the implementation was incorrect

evolvability refers to changes that are due to new features

portability refers to a system design that can move to new hardware or software platforms

interoperability refers to a system design that works well with other devices

Chapter 3 Terms

bottom-up approach is a form of system design where a design is constructed starting with simple components that are used to build small systems and those small systems are used to build bigger systems an so on

top-down approach is a form of system design where a design is described at the top-level as consisting of smaller subcomponents, which are described by still smaller subcomponents down to the smallest component

module is any self-contained operation that has: a name, an interface, and some functional description; and can refer to hardware, software or even something less concrete

Chapter 3 Terms

formal interface is the module's name and an enumeration of its operations including, for each operation, its inputs (if any), outputs (one or more), and name

general interface includes the formal interface and any additional protocol or implied communication (through shared, global memory for example)

implicit functional description of a module refers to the a module's description that is so universal that by convention it is simply understand how it functions, such as the module "FullAdder"

informal functional description of a module means its intended behavior is described in comments, exposition, or narrative

Chapter 3 Terms

formal functional description of a module means the behavior is either described mathematically (in terms of sets and functions) or otherwise codified, such as a completed C subroutine or a behavioral description in VHDL

implementation of a module is some realization of the module's intended functionality

instance of a module is a use of an implementation

instantiate means to make a copy of an instance

abstraction is the act or an instance of abstracting or taking away; an abstract or visionary idea

Chapter 3 Terms

state (of an application) is all of the data associated with an application's data and part of the system's storage; this includes data stored in registers, program counter, condition codes, RAM, configuration registers in I/O devices

cohesion is a means of measuring the abstraction, if the details inside of a module come together to implement an easily understood function, the module is said to have cohesion

coupling is a measure of how modules of a system are related to one another

refactoring is the task of looking at an existing design and rearranging the groupings and hierarchy without changing its functionality

Chapter 3 Terms

regression testing is a sometimes automated testing procedure that might be simulation driven (i.e. test benches) or it may be a set of systems that wraps around the component and exercises its functionality

software reference design is a piece of software where the functionally mimics the behavior of the whole system, even hardware modules that have not been implemented yet

control flow graph (CFG) is graph G = (V , E) where the vertices (or nodes) V of the graph are basic blocks and the directed edges indicate all possible paths that program could take at run-time

basic block is a maximal sequence of sequential instructions with a Single Entry and Single Exit (SESE) point

Chapter 3 Terms

system software refers to any software that assists the application, usually by adding a software interface to access the hardware

standalone C is a program that runs without the support of and any additional system software

cross-compiler is a high-level language translator that runs on one platform (a specific processor, a C library, and an operating system) but produces executable for another platform

Chapter 3 Terms

monitor is a primitive type of a debugger that is interrupt-driven, either the processor is interrupted or the application being debugged traps to the debugger; also, a monitor usually only supports the most basic functionality, reading/writing absolute addresses, setting breakpoints, manipulating registers

target see target machine

target machine in cross-compiler terms, the processor and operating environment for the embedded system

host see host machine

host machine (cross-compiling) when cross-compiling a program, the system that will run the compiler tools for a particular target is called a host machine