

Chapter 3

RTL Design with VHDL: From Requirements to Optimized Code

3.1 Prelude to Chapter

3.1.1 A Note on EDA for FPGAs and ASICs

The following is from John Cooley's column *The Industry Gadfly* from 2003/04/30. The title of this article is: "The FPGA EDA Slums".

For 2001, Dataquest reported that the ASIC market was US\$16.6 billion while the FPGA market was US\$2.6 billion.

What's more interesting is that the 2001 ASIC EDA market was US\$2.2 billion while the FPGA EDA market was US\$91.1 million. Nope, that's not a mistake. It's ASIC EDA and billion versus FPGA EDA and million. Do the math and you'll see that for every dollar spent on an ASIC project, roughly 12 cents of it goes to an EDA vendor. For every dollar spent on a FPGA project, roughly 3.4 cents goes to an EDA vendor. Not good.

It's the old free milk and a cow story according to Gary Smith, the Senior EDA Analyst at Dataquest. "Altera and Xilinx have fowled their own nest. Their free tools spoil the FPGA EDA market," says Gary. "EDA vendors know that there's no money to be made in FPGA tools."

3.2 Additional VHDL Features

3.2.1 Vectors

VHDL supports reading from and assigning to slices (aka "discrete subranges") of vectors. The rules for working with slices of vectors are listed below and illustrated in figure 3.1.

1. The ranges on both sides of the assignment must be the same.
2. The direction (downto or to) of each slice must match the direction of the signal declaration.
3. The direction of the target and expression may be different.

Declarations

```
-----
a, b      : in std_logic_vector(15 downto 0);
c, d, e    : out std_logic_vector(15 downto 0);
-----
ax, bx     : in std_logic_vector(0 to 15);
cx, dx, ex : out std_logic_vector(0 to 15);
-----
m, n       : in  unsigned(15 downto 0);
p, q, r    : out unsigned(15 downto 0);
-----
w, x       : in  signed(15 downto 0);
y, z       : out signed(15 downto 0);
-----
```

Legal code

```
c(3 downto 0) <= a(15 downto 12);
cx(0 to 3)   <= a(15 downto 12);
(e(3), e(4)) <= bx(12 to 13);
(e(5), e(6)) <= b(13 downto 12);
```

Illegal code

```
d(0 to 3)   <= a(15 to 12);           -- slice dirs must be same as decl
e(3) & e(2)  <= b(12 to 13);          -- syntax error on &
p(3 downto 0) <= (m + n)( 3 downto 0); -- syntax error on )(
z(3 downto 0) <= m(15 downto 12);     -- types on lhs and rhs must match
```

Figure 3.1: Illustration of Rules for Slices of Vectors

3.2.2 A Few More Miscellaneous VHDL Features

Some constructs that are useful and will be described in later chapters and sections:

report : print a message on stderr while simulating

assert : assertions about behaviour of signals, very useful with report statements.

generics : parameters to an entity that are defined at elaboration time.

attributes : predefined functions for different datatypes. For example: high and low indices of a vector.

3.3 RTL Coding Guidelines

This section gives guidelines for building robust, portable, and synthesizable VHDL code. Portability is both for different simulation and synthesis tools and for different implementation technologies.

Remember, there is a world of difference between getting a design to work in simulation and getting it to work on a real FPGA. And there is also a huge difference between getting a design to work in an FPGA for a few minutes of testing and getting thousands of products to work for months at a time in thousands of different environments around the world.

The coding guidelines here are designed both for helping you to get your E&CE 427 project to work as well as all of the subsequent industrial designs.

Finally, note that there are exceptions to every rule. You might find yourself in a circumstance where your particular situation (e.g. choice of tool, target technology, etc) would benefit from bending or breaking a guideline here. Within E&CE 427, of course, there won't be any such circumstances.

3.3.1 Signal Declarations

- Use signals, do **not** use variables

reason The intention of the creators of VHDL was for signals to be wires and variables to be just for simulation. Some synthesis tools allow some uses of variables, but when using variables, it is easy to create a design that works in simulation but not in real hardware.

- Use `std_logic` signals, do **not** use `bit` or `Boolean`

reason `std_logic` is the most commonly used signal type across synthesis tools, simulation tools, and cell libraries

- Use `in` or `out`, do **not** use `inout`

reason `inout` signals are tri-state.

note If you have an output signal that you also want to read from, you might be tempted to declare the mode of the signal to be `inout`. A better solution is to create a new, internal, signal that you both read from and write to. Then, your output signal can just read from the internal signal.

- Declare the primary inputs and outputs of chips as either `std_logic` and `std_logic_vector`. Do **not** use `signed` or `unsigned` for primary inputs or outputs.

reason Both the Altera tool Quartus and the Xilinx tool `ngd2vhdl` convert signed and unsigned vectors in entities into `std-logic-vectors`. If you want your same testbench to work for both functional simulation and timing simulation, you must **not** use signed or unsigned signals in the **top-level** entity of your chip.

note Signed and unsigned signals are fine inside testbenches, for non-top-level entities, and inside architectures. It is only the top-level entity that should not use signed or unsigned signals.

3.3.2 Flip-Flops and Latches

- Use flops, not latches (see section 1.8.2).
- Use D-flops, not T, JK, etc (see section 1.8.2).
- For every signal in your design, know whether it should be a flip-flop or combinational. Before simulating your design, examine the log file `LOG/dc_shell.log` to see if the flip flops in your circuit match your expectations, and to check that you don't have any latches in your design.

3.3.3 Inputs and Outputs

- Put flip flops on primary inputs and outputs of a chip

reason Creates more robust implementations. Signal delays between chips are unpredictable. Signal integrity can be a problem (remember transmission lines from E&CE 324?). Putting flip flops on inputs and outputs of chip provides clean boundaries between circuits.

note This only applies to primary inputs and outputs of a chip (the signals in the top-level entity). Within a chip, you should adopt a standard of putting flip-flops on either inputs or outputs of modules. Within a chip, you do not need to put flip-flops on both inputs and outputs.

3.3.4 Multiplexors and Tri-State Signals

- Use multiplexors, not tri-state buffers (see section 1.8.2).

3.3.5 Processes

- For a combinational process, the sensitivity list should contain all of the signals that are read in the process.

reason Gives consistent results across different tools. Many synthesis tools will implicitly include **all** signals that a process reads in its sensitivity list. This differs from the VHDL Standard. A tool that adheres to the standard will introduce latches if not all signals that are read from are included in the sensitivity list.

exception In a clocked process using an `if rising_edge`, it is acceptable to have only the clock in the sensitivity list

- For a combinational process, every signal that is assigned to, must be assigned to in every branch of if-then and case statements.

reason If a signal is not assigned a value in a path through a combinational process, then that signal will be a latch.

note For a clocked process, if a signal is not assigned a value in a clock cycle, then the flip-flop for that signal will have a chip-enable pin. Chip-enable pins are fine; they are available on flip-flops in essentially every cell library.

- Each signal should be assigned to in only one process.

reason Multiple processes driving the same signal is the same as having multiple gates driving the same wire. This can cause contention, short circuits, and other bad things.

exception Multiple drivers are acceptable for tri-state busses or if your implementation technology has wired-ANDs or wired-ORs. FPGAs don't have wired-ANDs or wired-ORs.

- Separate unrelated signals into different processes

reason Grouping assignments to unrelated signals into a single process can complicate the control circuitry for that process. Each branch in a case statement or if-then-else adds a multiplexor or chip-enable circuitry.

reason Synthesis tools generally optimize each process individually, the larger a process is, the longer it will take the synthesis program to optimize the process. Also, larger processes tend to be more complicated and can cause synthesis programs to miss helpful optimizations that they would notice in smaller processes.

3.3.6 State Machines

- In a state machine, illegal and unreachable states should transition to the reset state
 - reason** Creates more robust implementations. In the field, your circuit will be subjected to illegal inputs, voltage spikes, temperature fluctuations, clock speed variations, etc. At some point in time, something weird will happen that will cause it to jump into an illegal state. Having a system reset and reboot is much better than having it generate incorrect outputs that aren't detected.
- If your state machine has less than 16 states, use a one-hot encoding.
 - reason** For n states, a one-hot encoding uses n flip-flops, while a binary encoding uses $\log_2 n$ flip-flops. One-hot signals are simpler to decode, because only one bit must be checked to determine if the circuit is in a particular state. For small values of n , a one-hot signal results in a smaller and faster circuit. For large values of n , the number of signals required for a one-hot design is too great of a penalty to compensate for the simplicity of the decoding circuitry.
 - note** Using an enumerated type for states allows the synthesis tool to choose state encodings that it thinks will work well to balance area and clock speed. Quartus uses a "modified one-hot" encoding, where the bit that denotes the reset state is inverted. That is, when the reset bit is '0', the system is in the reset state and when the reset bit is a '1' the system is not in the reset state. The other bits have the normal polarity. The result is that when the system is in the reset state, all bits are '0' and when the system is in a non-reset state, two bits are '1'.
 - note** Using your own encoding allows you to leverage knowledge about your design that the synthesis tool might not be able to deduce.

3.3.7 Reset

- Include a reset signal in all clocked circuits.
 - reason** For most implementation technologies, when you power-up the circuit, you do not know what state it will start in. You need a reset signal to get the circuit into a known state.
 - reason** If something goes wrong while the circuit is running, you need a way to get it into a known state.
- For implicit state machines (section 3.7.1.3), check for reset after every wait statement.
 - reason** Missing a wait statement means that your circuit might not notice a reset signal, or different signals could reset in different clock cycles, causing your circuit to get out of synch.
- Connect reset to the important control signals in the design, such as the state signal. Do not reset every flip flop.
 - reason** Using reset adds area and delay to a circuit. The fewer signals that need reset, the faster and smaller your design will be.
 - note** Connect the reset signal to critical flip-flops, such as the state signal. Datapath signals rarely need to be reset. You do not need to reset every signal
- Use **synchronous**, not asynchronous, reset
 - reason** Creates more robust implementations. Signal propagation delays mean that asynchronous resets cause different parts of the circuit to be reset at different times. This can lead to glitches, which then might cause the circuit to move to an illegal state.

Covering All Cases

When writing case statements or selected assignments that test the value of `std_logic` signals, you will get an error unless you include a provision for non '1'/'0' signals.

For example:

```
signal t : std_logic;
...
case t is
  when '1' => ...
  when '0' => ...
end case;
```

will result in an error message about missing cases. You must provide for `t` being 'H', 'U', etc. The simplest thing to do is to make the last test when `other`.

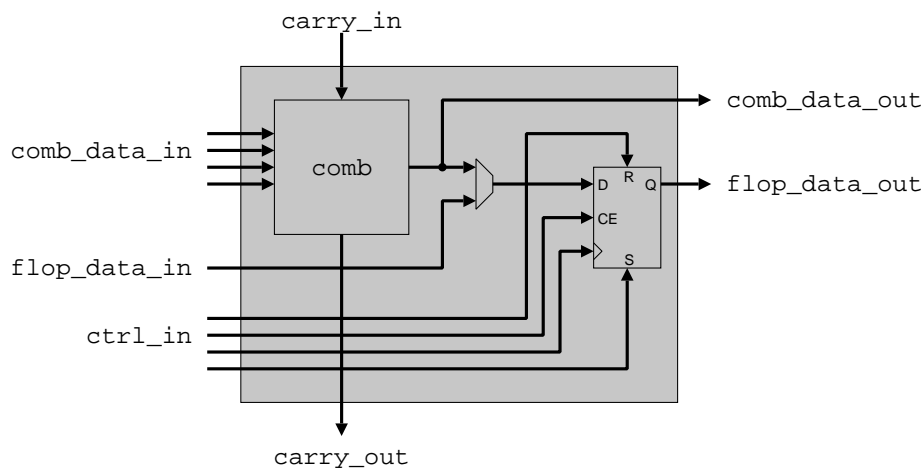
3.4 FPGA Background and Coding Guidelines

3.4.1 Generic FPGA Hardware

3.4.1.1 Generic FPGA Cell

“Cell” = “Logic Element” (LE) in Altera
 = “Configurable Logic Block” (CLB) in Xilinx

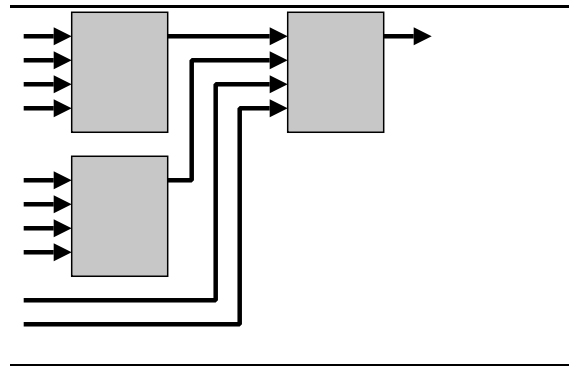
Supplemental material available online (fpga)



3.4.1.2 Area Estimation

For FPGAs, you can estimate the area of a design by counting the number of flip-flops in the fanin of each flip-flop. Only flip-flops count, because combinational signals are collapsed into the circuitry within an FPGA cell. The circuitry for any flip-flop signal with up to four source flip-flops can be implemented on a single FPGA cell. If a flip-flop signal is dependent upon five source flip-flops, then two FPGA cells are required.

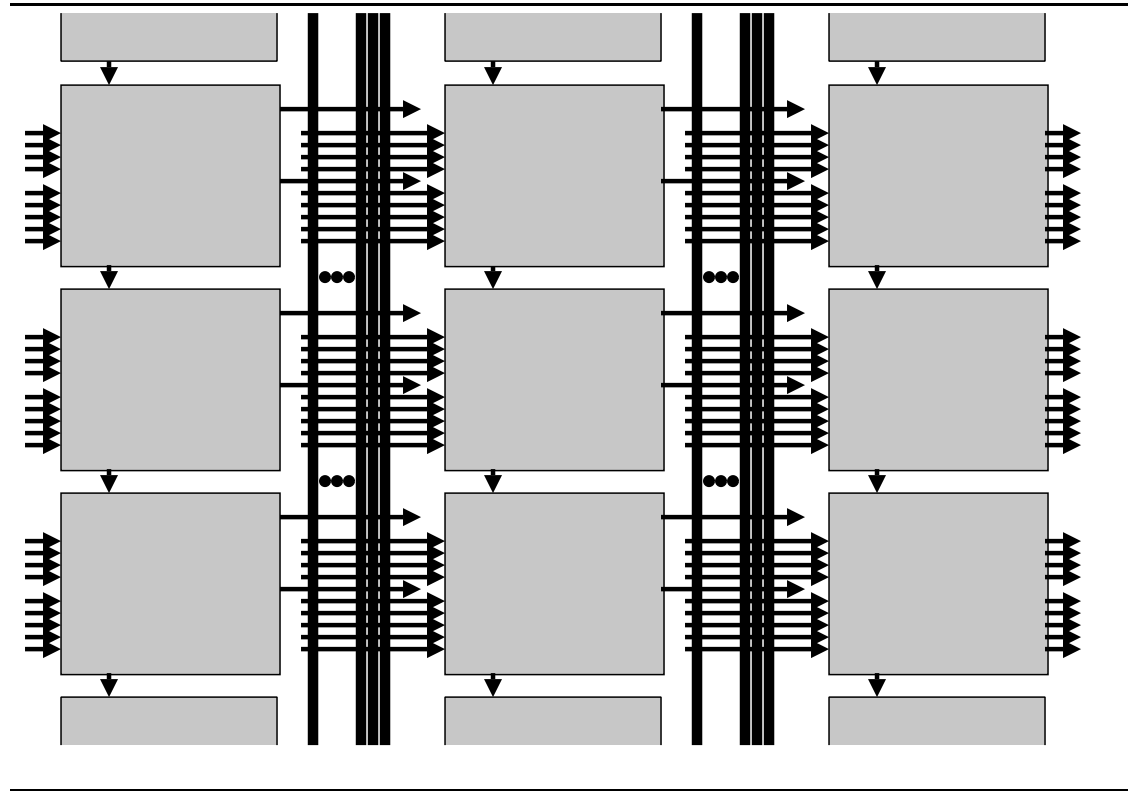
Source flops	Cells
1	1
2	1
3	1
4	1
5	2
6	2
7	2
8	3
9	3
10	3
11	4



This technique is generally an overestimate, because a single cell can drive several other cells (common subexpression elimination).

3.4.1.3 Interconnect for Generic FPGA

NB: In these slides, the space between tightly grouped wires sometimes disappears, making a group of wires appear to be a single large wire.

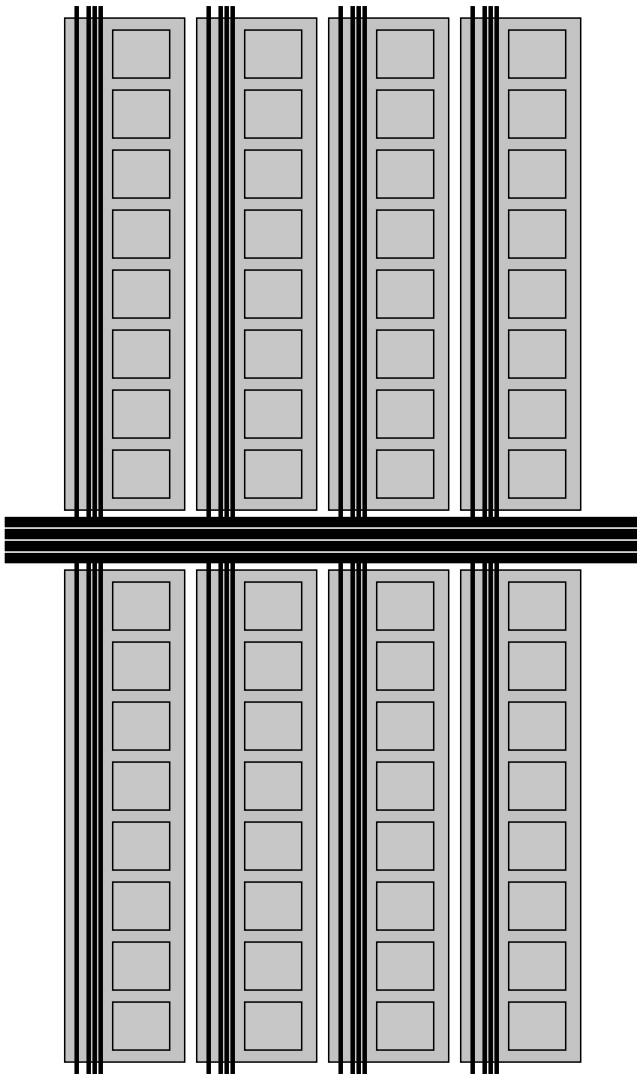


There are two types of wires that connect a cell to the rest of the chip:

- General purpose interconnect (configurable, slow)
- Carry chains and cascade chains (vertically adjacent cells, fast)

Supplemental material available online (connect)

3.4.1.4 Blocks of Cells for Generic FPGA



Cells are organized into blocks. There is a great deal of interconnect (wires) between cells within a single block. In large FPGAs, the blocks are organized into larger blocks. These large blocks might themselves be organized into even larger blocks. Think of an FPGA as bunch of nested `for-generate` statements that replicate a single component (cell) hundreds of thousands of times.

Cells not used for computation can be used as “wires” to shorten length of path between cells.

3.4.1.5 Clocks for Generic FPGAs

Characteristics of clock signals:

- High fanout (drive many gates)
- Long wires (destination gates scattered all over chip)

Characteristics of FPGAs:

- Very few gates that are large (strong) enough to support a high fanout.
- Very few wires that traverse entire chip and can be connected to every flip-flop.

3.4.1.6 Special Circuitry in FPGAs

Memory

For five or more years, FPGAs have had special circuits for RAM and ROM. In Altera FPGAs, these circuits are called ESBs (Embedded System Blocks). These special circuits are possible because many FPGAs are fabricated on the same processes as SRAM chips. So, the FPGAs simply contain small chunks of SRAM.

Microprocessors

A new feature to appear in FPGAs in 2001 and 2002 is hardwired microprocessors on the same chip as programmable hardware.

	Hard	Soft
Altera	Arm 922T with 200 MIPs	Nios with ?? MIPs
Xilinx: Virtex-II Pro	Power PC 405 with 420 D-MIPs	Microblaze with 100 D-MIPs

The Xilinx-II Pro has 4 Power PCs and enough programmable hardware to implement the first-generation Intel Pentium microprocessor.

Arithmetic Circuitry

A new feature to appear in FPGAs in 2001 and 2002 is hardwired circuits for multipliers and adders.

Altera: Mercury	16 × 16 at 130MHz
Xilinx: Virtex-II Pro	18 × 18 at ???MHz

Using these resources can improve significantly both the area and performance of a design.

Input / Output

Recently, high-end FPGAs have started to include special circuits to increase the bandwidth of communication with the outside world.

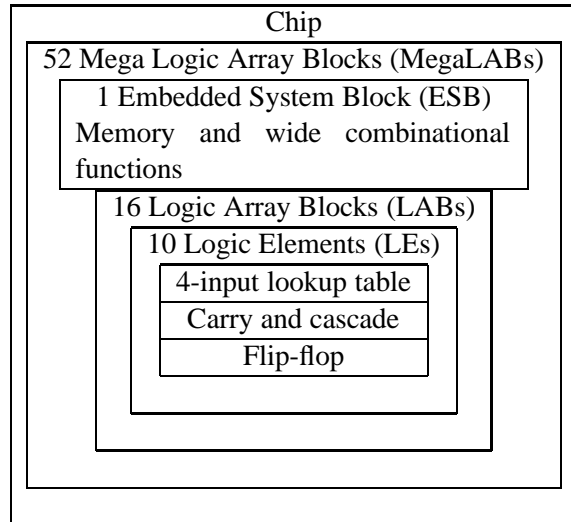
	Product
Altera	True-LVDS (1 Gbps)
Xilinx	Rocket I/O (3 Gbps)

3.4.2 Generic-FPGA Coding Guidelines

- Flip-flops are almost free in FPGAs
 - reason** In FPGAs, the area consumed by a design is usually determined by the amount of combinational circuitry, not by the number of flip-flops.
- Aim for using 80–90% of the cells on a chip.
 - reason** If you use more than 90% of the cells on a chip, then the place-and-route program might not be able to route the wires to connect the cells.
 - reason** If you use less than 80% of the cells, then probably:
 - there are optimizations that will increase performance and still allow the design to fit on the chip;
 - or** you spent too much human effort on optimizing for low area;
 - or** you could use a smaller (cheaper!) chip.
 - exception** In E&CE 427 (unlike in real life), the mark is based on the actual number of cells used.
- Use just one clock signal
 - reason** If all flip-flops use the same clock, then the clock does not impose any constraints on where the place-and-route tool puts flip-flops and gates. If different flip-flops used different clocks, then flip-flops that are near each other would probably be required to use the same clock.
- Use only one edge of the clock signal
 - reason** There are two ways to use both rising and falling edges of a clock signal: have rising-edge and falling-edge flip flops, or have two different clock signals that are inverses of each other. Most FPGAs have only rising-edge flip flops. Thus, using both edges of a clock signal is equivalent to having two different clock signals, which is deprecated by the preceding guideline.

3.4.3 Altera APEX20K Information and Coding Guidelines

APEX20K Block Hierarchy



Each level of hierarchy has its own interconnect (wires).

LE Interconnect

LE Computation and Storage

- 4-input lookup table (LUT)
- Carry-chain computation circuitry
- Cascade-chain computation circuitry
- Flip-flop with load, clear, clock-enable

- 4 data inputs
- 2 data outputs
- Carry in, carry out
- Cascade in, cascade out
- Clock, clock-enable
- Async clear, synch set (load), synch clear (reset)
- Global reset

Initialization

The Altera APEX20K chips initialize all flip flops to '0' at startup. To mimic this behaviour in simulation, you should put an initial value of '0' on all flip flops. If you are doing your own encoding for a state machine, choose the reset state to be encoded as all zeroes.

You should **not** put initial values on inputs or combinational signals.

3.5 Design Flow

3.5.1 Generic Design Flow

Most people agree on the general terminology and process for a digital hardware design flow. However, each book and course has its own particular way of presenting the ideas. Here we will lay out the consistent set of definitions that we will use in E&CE 427. This might be different from what you have seen in other courses or on a work term. Focus on the ideas and you will be fine both now and in the future.

The design flow presented here focuses on the *artifacts* that we work with, rather than the operations that are performed on the artifacts. This is because the same operations can be performed at different points in the design flow, while the artifacts each have a unique purpose.

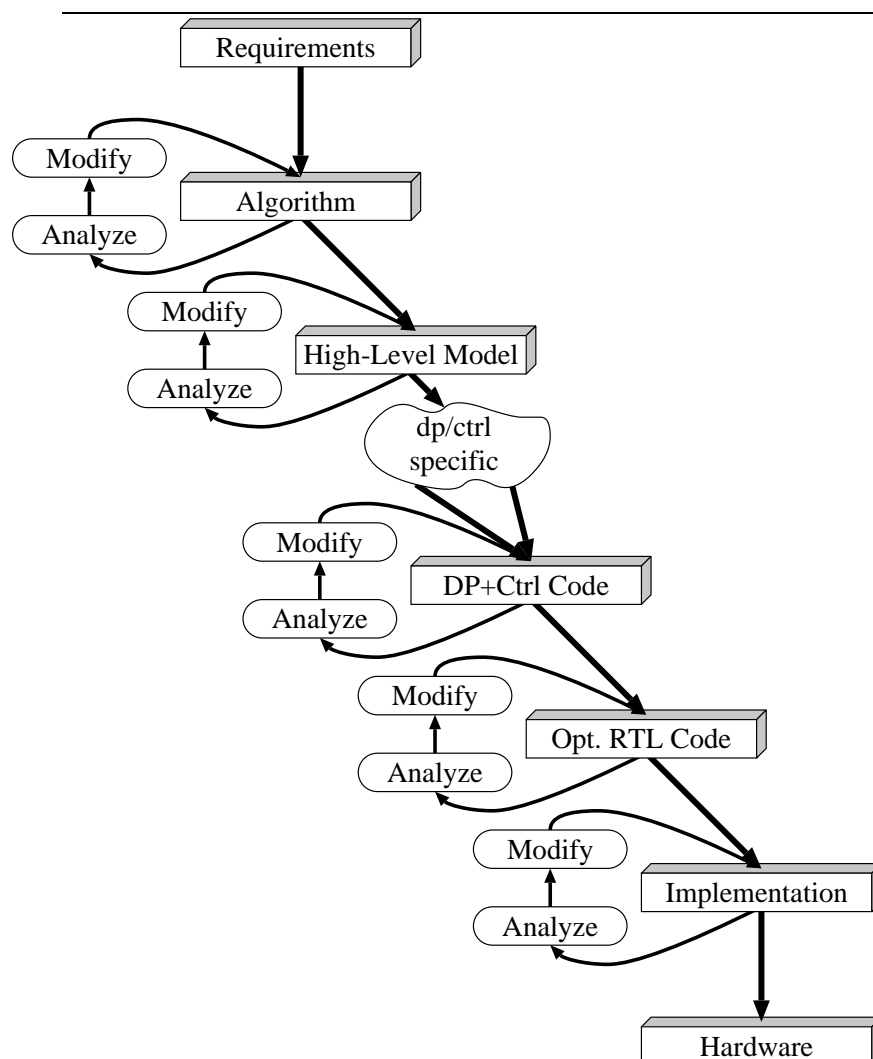


Figure 3.2: Generic Design Flow

Table 3.1: Artifacts in the Design Flow

Requirements	Description of what the customer wants
Algorithm	Functional description of computation. Probably not synthesizable. Could be a flowchart, software, diagram, mathematical equation, <i>etc.</i> .
High-Level Model	HDL code that is not necessarily synthesizable, but divides algorithm into signals and clock cycles. Possibly mixes datapath and control. In VHDL, could be a single process that captures the behaviour of the algorithm. Usually synthesizable; resulting hardware is usually big and slow compared to optimized RTL code.
Dataflow Diagram	A picture that depicts the datapath computation over time, clock-cycle by clock-cycle (Section 3.8)
Hardware Block Diagram	A picture that depicts the structure of the datapath: the components and the connections between the components. (e.g., netlist or schematic)
State Machine	A picture that depicts the behaviour of the control circuitry over time (Section 3.7)
DP+Ctrl RTL code	Synthesizable HDL code that separates the datapath and control into separate processes and assignments.
Optimized RTL Code	HDL code that has been written to meet design goals (high performance, low power, small, etc.)
Implementation Code	A collection of files that include all of the information needed to build the circuit: HDL program targeted for a particular implementation technology (e.g. a specific FPGA chip), constraint files, script files, etc.

Recommendation: *Spend the time up front to plan a good design on paper. Use dataflow diagrams and state machines to predict performance and area. The E&CE 427 project might appear to be sufficiently small and simple that you can go straight to RTL code. However, you will probably produce a more optimal design with less effort if you explore high-level optimizations with dataflow diagrams and state machines.*

3.5.2 Implementation Flows

Synopsys Design Compiler and FPGA Compiler are general-purpose synthesis programs. They have very few, if any, technology-specific algorithms. Instead, they rely on libraries to describe technology-specific parameters of the primitive building blocks (e.g. the delay and area of individual gates, PLAs, CLBs, flops, memory arrays).

Mentor Graphic's product Leonardo Spectrum, Cadence's product BuildGates, and Synplicity's product Synplify are similar. In comparison, Avant! (Now owned by Synopsys) and Cadence sell separate tools that do place-and-route and other low-level (physical design) tasks.

These general-purpose synthesis tools do not (generally) do the final stages of the design, such as place-and-route and timing analysis, which are very specific to a given implementation technology. The implementation-

technology-specific tools generally also produce a VHDL file that accurately models the chip. We will refer to this file as the “**implementation VHDL code**”.

With Synopsys and the Altera tool Quartus, we compile the VHDL code into an EDIF file for the netlist and a TCL file for the commands to Quartus. Quartus then generates a `sof` (SRAM Object File), which can be downloaded to an Altera SRAM-based FPGA. The extension of the implementation VHDL file is often `.vho`, for “VHDL output”.

With the Synopsys and Xilinx tools, we compile VHDL code into a Xilinx-specific design file (`xnf` — Xilinx netlist file). We then use the Xilinx tools to generate a `bit` file, which can be downloaded to a Xilinx FPGA. The name of the implementation VHDL file is often suffixed with `routed.vhd`.

Terminology: “Behavioural” and “Structural”

NOTE: behavioural and structural models *The phrases “behavioural model” and “structural model” are commonly used for what we’ll call “high-level models” and “synthesizable models”. In most cases, what people call structural code contains both structural and behavioural code. The technically correct definition of a structural model is an HDL program that contains only component instantiations and generate statements. Thus, even a program with `c <= a AND b;` is, strictly speaking, behavioural.*

3.5.3 Design Flow: Datapath vs Control vs Storage

3.5.3.1 Classes of Hardware

Each circuit tends to be dominated by either its datapath, control (state machine) or storage (memory).

- Datapath
 - Purpose: compute output data based on input data
 - Each “parcel” of input produces one “parcel” of output
 - Examples: arithmetic, decoders
- Storage
 - Purpose: hold data for future use
 - Data is not modified while stored
 - Examples: register files, FIFO queues
- Control
 - Purpose: modify internal state based on inputs, compute outputs from state and inputs
 - Mostly individual signals, few data (vectors)
 - Examples: bus arbiters, memory-controllers

All three classes of circuits (datapath, control, and storage) follow the same generic design flow (Figure 3.2) and use dataflow diagrams, hardware block diagrams, and state machines. The differences in the design flows appear in the relative amount of effort spent on each type of description and the order in which the different descriptions are used. The differences are most pronounced in the transition from the high-level model to the model that separates the datapath and control circuitry.

3.5.3.2 Datapath-Centric Design Flow

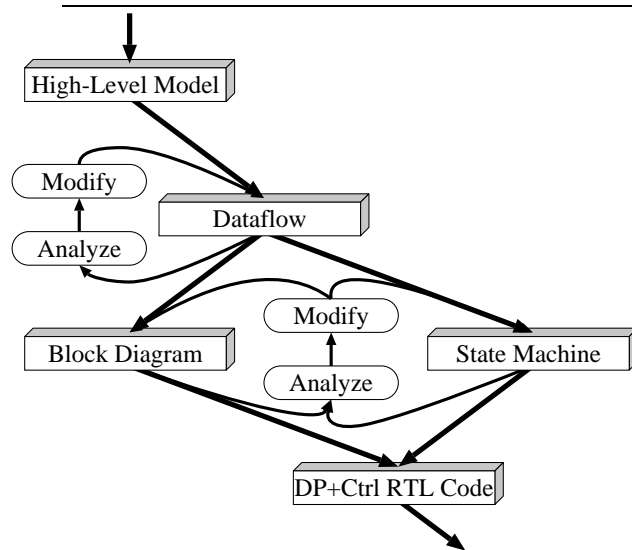


Figure 3.3: Datapath-Centric Design Flow

3.5.3.3 Control-Centric Design Flow

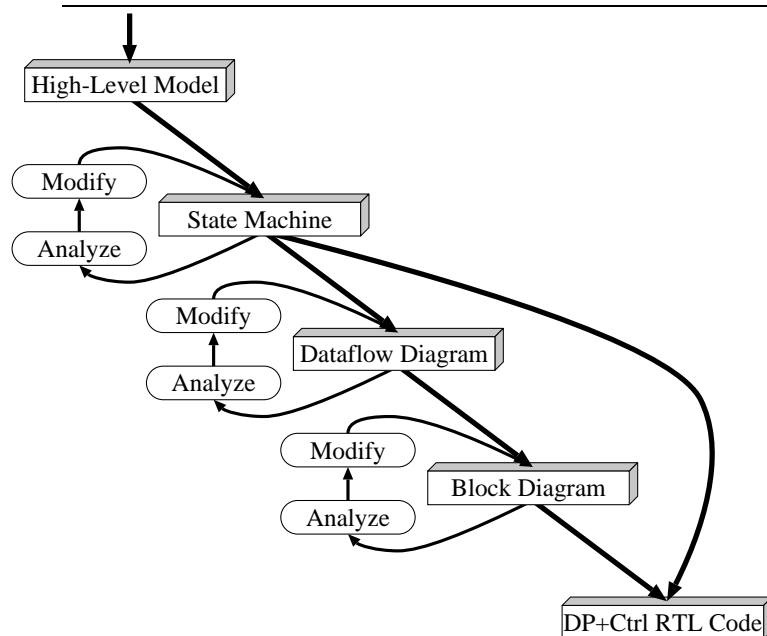


Figure 3.4: Control-Centric Design Flow

3.5.3.4 Storage-Centric Design Flow

In E&CE 427, we won't be discussing storage-centric design. Storage-centric design differs from datapath- and control-centric design in that storage-centric design focusses on building many replicated copies of small cells.

Storage-centric designs include a wide range of circuits, from simple memory arrays to complicated circuits such as register files, translation lookaside buffers, and caches. The complicated circuits can contain large and very intricate state machines, which would benefit from some of the techniques for control-centric circuits.

3.6 Algorithms and High-Level Models

For designs with significant control flow, algorithms can be described in software languages, flow-charts, abstract state machines, algorithmic state machines, etc.

For designs with trivial control flow (e.g. every parcel of input data undergoes the same computation), data-dependency graphs (section 3.6.2) are a good way to describe the algorithm.

For designs with a small amount of control flow (e.g. a microprocessor, where a single decision is made based upon the opcode) a set of data-dependency graphs is often a good choice.

Software executes in series; hardware executes in parallel

When creating an algorithmic description of your hardware design, think about how you can represent parallelism in the algorithmic notation that you are using, and how you can exploit parallelism to improve the performance of your design.

3.6.1 Flow Charts and State Machines

Flow charts and various flavours of state machines are covered well in many courses. Generally everything that you've learned about these forms of description are also applicable in hardware design.

In addition, you can exploit parallelism in state machine design to create *communicating finite state machines*. A single complex state machine can be factored into multiple simple state machines that operate in parallel and communicate with each other.

3.6.2 Data-Dependency Graphs

In software, the expression: $((((a + b) + c) + d) + e) + f$ takes the same amount of time to execute as: $(a + b) + (c + d) + (e + f)$.

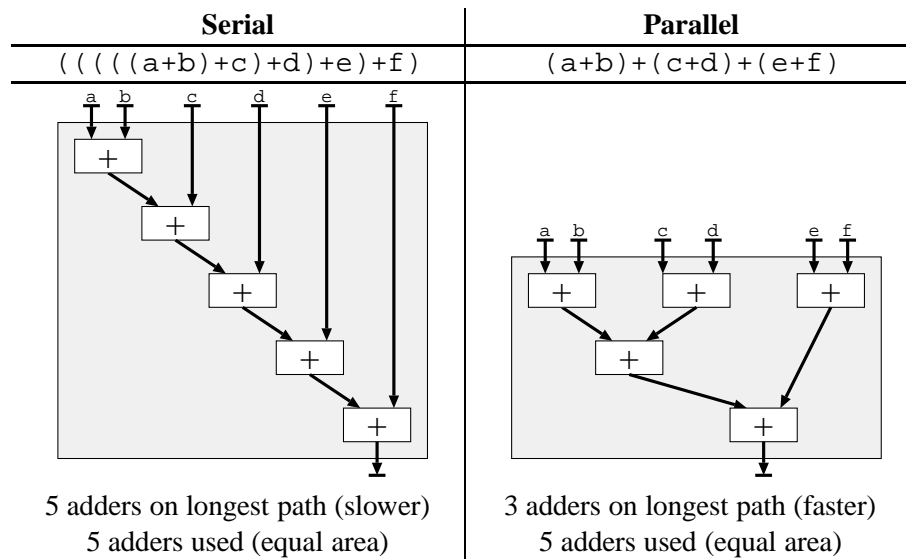
But, remember: hardware runs in parallel. In algorithmic descriptions, parentheses can guide parallel vs serial execution.

Datadependency graphs capture algorithms of datapath-centric designs.

Datapath-centric designs have few, if any, control decisions: every parcel of input data undergoes the same computation.

Serial vs Parallel

Supplemental material available online (ser-par)



3.6.3 High-Level Models

There are many different types of high-level models, depending upon the purpose of the model and the characteristics of the design that the model describes. Some models may capture power consumption, others performance, others data functionality.

High-level models are used to estimate the most important design metrics very early in the design cycle. If power consumption is more important than performance, then you might write high-level models that can predict the power consumption of different design choices, but which has no information about the number of clock cycles that a computation takes, or which predicts the latency inaccurately. Conversely, if performance is important, you might write clock-cycle accurate high-level models that do not contain any information about power consumption.

Conventionally, performance has been the primary design metric. Hence, high-level models that predict performance are more prevalent and more well understood than other types of high-level models. There are many research and entrepreneurial opportunities for people who can develop tools and/or languages for high-level models for estimating power, area, maximum clock speed, etc.

In E&CE 427 we will limit ourselves to the well-understood area of high-level models for performance prediction.

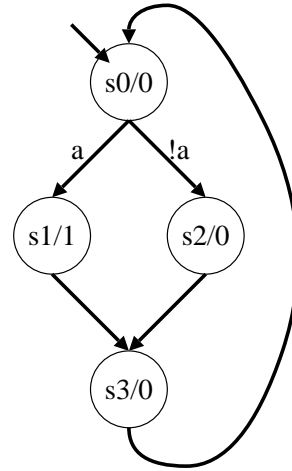
3.7 Finite State Machines in VHDL

3.7.1 Introduction to State-Machine Design

3.7.1.1 Mealy vs Moore State Machines

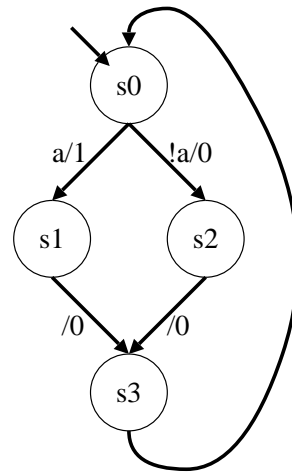
Moore Machines

- Outputs are dependent upon only the state
- No combinational paths from inputs to outputs



Mealy Machines

- Outputs are dependent upon both the state and the inputs
- Combinational paths from inputs to outputs



3.7.1.2 Introduction to State Machines and VHDL

A state machine is generally written as a single clocked process, or as a pair of processes, where one is clocked and one is combinational.

Design Decisions

- Moore vs Mealy (Sections 3.7.2 and 3.7.3)
- Implicit vs Explicit (Section 3.7.1.3)
- State values in explicit state machines: Enumerated type vs constants (Section 3.7.6.1)
- State values for constants: encoding scheme (binary, gray, one-hot, ...) (Section 3.7.6.2)

VHDL Constructs for State Machines

The following VHDL control constructs are useful to steer the transition from state to state:

- | | |
|------------------------|--------|
| • if ... then ... else | • loop |
| • case | • next |
| • for ... loop | • exit |
| • while ... loop | |

3.7.1.3 Explicit vs Implicit State Machines

There are two broad styles of writing state machines in VHDL: explicit and implicit. “Explicit” and “implicit” refer to whether there is an explicit state signal in the VHDL code. Explicit state machines have a state signal in the VHDL code. Implicit state machines do not contain a state signal. Instead, they use VHDL processes with multiple wait statements to control the execution.

In the explicit style of writing state machines, each process has at most one wait statement. For the explicit style of writing state machines, there are two sub-categories: “current state” and “current+next state”.

In the explicit-current style of writing state machines, the state signal represents the current state of the machine and the signal is assigned its next value in a clocked process.

In the explicit-current+next style, there is a signal for the current state and another signal for the next state. The next-state signal is assigned its value in a combinational process or concurrent statement and is dependent upon the current state and the inputs. The current-state signal is assigned its value in a clocked process and is just a flopped copy of the next-state signal.

For the implicit style of writing state machines, the synthesis program adds an implicit register to hold the state signal and combinational circuitry to update the state signal. In Synopsys synthesis tools, the state signal defined by the synthesizer is named `multiple_wait_state_reg`.

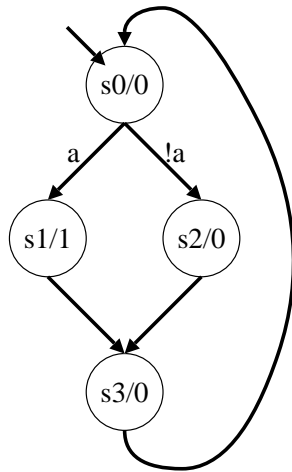
We can think of implicit state machines as having “0 state signals”, explicit-current state machines as having “1 state signal”, and explicit-current+next state machines as having “2 state signals”.

As with all topics in E&CE 427, there are tradeoffs between these different styles of writing state machines. Most books teach only the explicit-current+next style. This style is the style closest to the hardware, which means that they are more amenable to optimization through human intervention, rather than relying on a synthesis tool for optimization. The advantage of the implicit style is that they are concise and readable for control flows consisting of nested loops and branches (e.g. the type of control flow that appears in

software). For control flows that have less structure, it can be difficult to write an implicit state machine. Very few books or synthesis manuals describe multiple-wait statement processes, but they are relatively well supported among synthesis tools.

NB: The terminology of “explicit” and “implicit” is somewhat standard, in that some descriptions of processes with multiple wait statements describe the processes as having “implicit state machines”. There is no standard terminology to distinguish between the two explicit styles: explicit-current+next and explicit-current.

3.7.2 Implementing a Simple Moore Machine

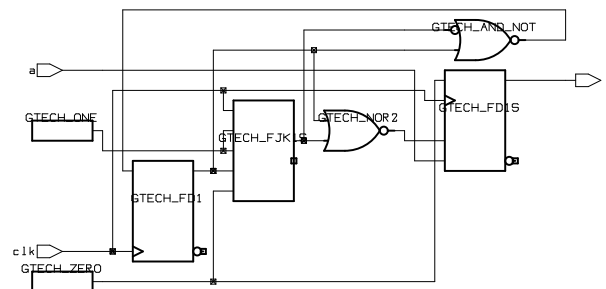


```
entity simple is
  port (
    a, clk : in std_logic;
    z : out std_logic
  );
end simple;
```

3.7.2.1 Implicit Moore State Machine

```
architecture moore_implicit of simple is
begin
    process
    begin
        z <= '0';
        wait until rising_edge(clk);
        if (a = '1') then
            z <= '1';
        else
            z <= '0';
        end if;
        wait until rising_edge(clk);
        z <= '0';
        wait until rising_edge(clk);
    end process;
end moore_implicit;
```

Flops	
Gates	
Delay	
Tokens	



design: noore	designer:	date: 9/25/2002
technology: gtech	company:	sheet: 1 of 1

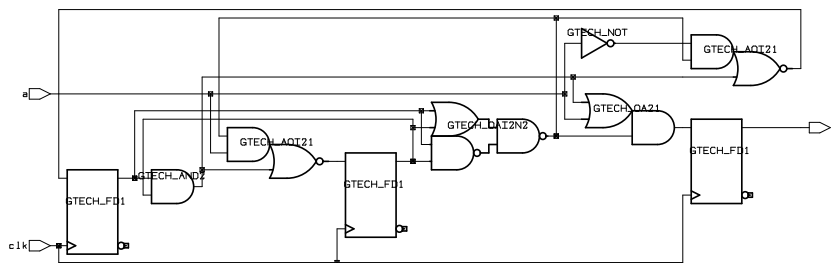
3.7.2.2 Explicit Moore with Flopped Output

```

architecture moore_explicit_v1 of simple is
    type state_ty is (s0, s1, s2, s3);
    signal state : state_ty;
begin
    process (clk)
    begin
        if rising_edge(clk) then
            case state is
                when s0 =>
                    if (a = '1') then
                        state <= s1;
                        z    <= '1';
                    else
                        state <= s2;
                        z    <= '0';
                    end if;
                when s1 | s2 =>
                    state <= s3;
                    z    <= '0';
                when s3 =>
                    state <= s0;
                    z    <= '1';
            end case;
        end if;
    end process;
end moore_explicit_v1;

```

Flops	
Gates	
Delay	
Tokens	



design: moore_v2	designer:	date: 9/25/2002
technology: gtech	company:	sheet: 1 of 1

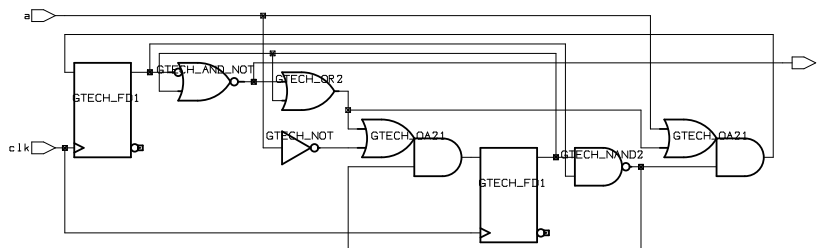
3.7.2.3 Explicit Moore with Combinational Outputs

```

architecture moore_explicit_v2 of simple is
    type state_ty is (s0, s1, s2, s3);
    signal state : state_ty;
begin
    process (clk)
    begin
        if rising_edge(clk) then
            case state is
                when s0 =>
                    if (a = '1') then
                        state <= s1;
                    else
                        state <= s2;
                    end if;
                when s1 | s2 =>
                    state <= s3;
                when s3 =>
                    state <= s0;
            end case;
        end if;
    end process;
    z <= '1' when (state = s1)
        else '0';
end moore_explicit_v2;

```

Flops	
Gates	
Delay	
Tokens	



design: moore_v3	designer:	date: 9/25/2002
technology: gtech	company:	sheet: 1 of 1

3.7.2.4 Explicit-Current+Next Moore with Concurrent Assignment

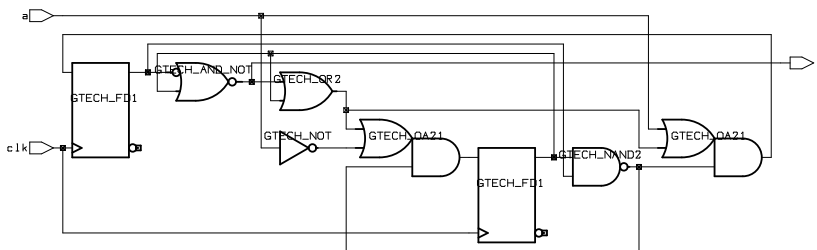
```

architecture moore_explicit_v3 of simple is
    type state_ty is (s0, s1, s2, s3);
    signal state, state_nxt : state_ty;
begin
    process (clk)
    begin
        if rising_edge(clk) then
            state <= state_nxt;
        end if;
    end process;
    state_nxt <=  s1 when (state = s0) and (a = '1')
                else s2 when (state = s0) and (a = '0')
                else s3 when (state = s1) or  (state = s2)
                else s0;
    z <=  '1' when (state = s1)
        else '0';
end moore_explicit_v3;

```

Flops	
Gates	
Delay	
Tokens	

The hardware synthesized from this architecture is the same as that synthesized from `moore_explicit_v2`, which is written in the current-explicit style.



design: moore_v4	designer:	date: 9/25/2002
technology: gtech	company:	sheet: 1 of 1

3.7.2.5 Explicit-Current+Next Moore with Combinational Process

```

architecture moore_explicit_v4 of simple is
    type state_ty is (s0, s1, s2, s3);
    signal state, state_nxt : state_ty;
begin
    process (clk)
    begin
        if rising_edge(clk) then
            state <= state_nxt;
        end if;
    end process;
    process (state, a)
    begin
        case state is
            when s0 =>
                if (a = '1') then
                    state_nxt <= s1;
                else
                    state_nxt <= s2;
                end if;
            when s1 | s2 =>
                state_nxt <= s3;
            when s3 =>
                state_nxt <= s0;
        end case;
    end process;
    z <= '1' when (state = s1)
        else '0';
end moore_explicit_v4;

```

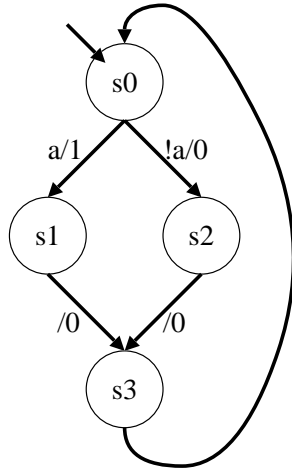
For this architecture, we change the selected assignment to `state` into a combinational process using a case statement.

Flops	
Gates	
Delay	
Tokens	

The hardware synthesized from this architecture is the same as that synthesized from `moore_explicit_v2` and `v3`.

3.7.3 Implementing a Simple Mealy Machine

This is the same entity as for the simple Moore state machine. The behaviour of the Mealy machine is the same as the Moore machine, except for the timing relationship between the output (z) and the input (a).



```
entity simple is
  port (
    a, clk : in std_logic;
    z : out std_logic
  );
end simple;
```

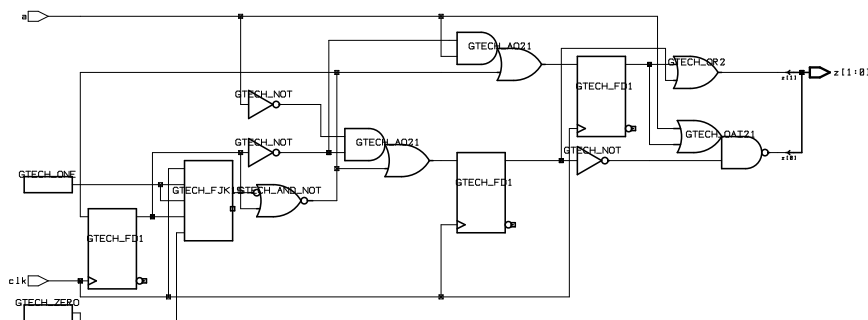
3.7.3.1 Implicit Mealy State Machine

```

architecture implicit_mealy of simple is
  type state_ty is (s0, s1, s2, s3);
  signal state : state_ty;
begin
  process
  begin
    state <= s0;
    wait until rising_edge(clk);
    if (a = '1') then
      state <= s1;
    else
      state <= s2;
    end if;
    wait until rising_edge(clk);
    state <= s3;
    wait until rising_edge(clk);
  end process;
  z <= '1' when (state = s0) and a = '1'
    else '0';
end mealy_implicit;

```

Flops
Gates
Delay
Tokens



design: mealy	designer:	date: 9/25/2002
technology: gtech	company:	sheet: 1 of 1

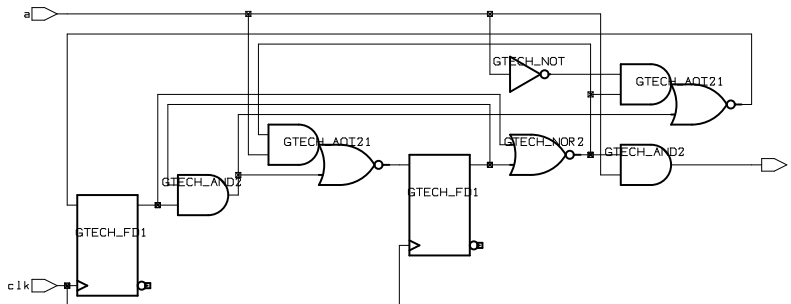
3.7.3.2 Explicit Mealy State Machine

```

architecture mealy_explicit of simple is
    type state_ty is (s0, s1, s2, s3);
    signal state : state_ty;
begin
    process (clk)
    begin
        if rising_edge(clk) then
            case state is
                when s0 =>
                    if (a = '1') then
                        state <= s1;
                    else
                        state <= s2;
                    end if;
                when s1 | s2 =>
                    state <= s3;
                when others =>
                    state <= s0;
            end case;
        end if;
    end process;
    z <= '1' when (state = s0) and a = '1'
        else '0';
end mealy_explicit;

```

Flops
Gates
Delay
Tokens



design: nealy_v2	designer:	date: 9/26/2002
technology: gtech	company:	sheet: 1 of 1

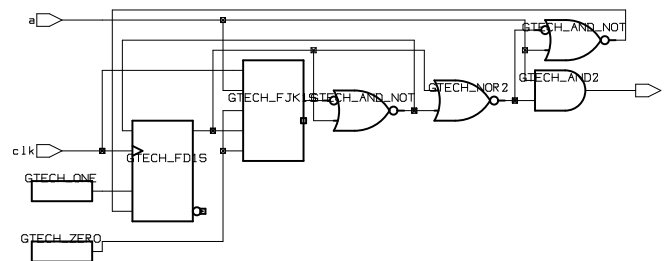
3.7.3.3 Explicit-Current+Next Mealy

architecture mealy_explicit_v2 of simple is

```

    type state_ty is (s0, s1, s2, s3);
    signal state, state_nxt : state_ty;
begin
    process (clk)
    begin
        if rising_edge(clk) then
            state <= state_nxt;
        end if;
    end process;
    state_nxt <=  s1 when (state = s0) and a = '1'
                  else s2 when (state = s0) and a = '0'
                  else s3 when (state = s1) or (state = s2)
                  else s0;
    z <=  '1' when (state = s0) and a = '1'
          else '0';
end mealy_explicit_v2;
```

Flops
Gates
Delay
Tokens



design: mealy_v3	designer:	date: 9/26/2002
technology: gtech	company:	sheet: 1 of 1

3.7.4 Reset

All circuits should have a reset signal that puts the circuit back into a good initial state.

There are standard ways to add a reset signal to both explicit and implicit state machines.

It is important that reset is tested on **every** clock cycle, otherwise a reset might not be noticed, or your circuit will be slow to react to reset and could generate illegal outputs after reset is asserted.

Reset with Implicit State Machine

With an implicit state machine, we need to insert a loop in the process and test for reset after each wait statement.

Here is the implicit Moore machine from section 3.7.2.1 with reset code added in bold.

```
architecture moore_implicit of simple is
begin
  process
  begin
    init : loop                                -- outermost loop
      z <= '0';
      wait until rising_edge(clk);
      next init when (reset = '1'); -- test for reset
      if (a = '1') then
        z <= '1';
      else
        z <= '0';
      end if;
      wait until rising_edge(clk);
      next init when (reset = '1'); -- test for reset
      z <= '0';
      wait until rising_edge(clk);
      next init when (reset = '1'); -- test for reset
    end process;
  end moore_implicit;
```

Reset with Explicit State Machine

Reset is often easier to include in an explicit state machine, because we need only put a test for `reset = '1'` in the clocked process for the state.

The pattern for an explicit-current style of machine is:

```
process (clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      state <= S0;
    else
      if ... then
        state <= ...;
      elsif ... then
        ... -- more tests and assignments to state
      end if;
    end if;
  end if;
end process;
```

Applying this pattern to the explicit Moore machine from section 3.7.2.3 produces:

architecture moore_explicit_v2 of simple is

```
  type state_ty is (s0, s1, s2, s3);
  signal state : state_ty;
```

begin

```
  process (clk)
  begin
    if rising_edge(clk) then
      if (reset = '1') then
        state <= s0;
      else
        case state is
          when s0 =>
            if (a = '1') then
              state <= s1;
            else
              state <= s2;
            end if;
          when s1 | s2 =>
            state <= s3;
          when s3 =>
            state <= s0;
          end case;
        end if;
      end if;
    end process;
```

```
  z <= '1' when (state = s1)
    else '0';
```

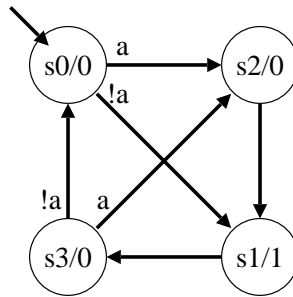
end moore_explicit_v2;

The pattern for an explicit-current+next style is:

```
process (clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      state_cur <= reset_state;
    else
      state_cur <= state_nxt;
    end if;
  end if;
end process;
```

3.7.5 Implicit vs Explicit Revisited

Because implicit state machines are written with loops, if-then-elses, cases, etc. it is difficult to write in an implicit style some state machines with complicated control flows. The following example illustrates the point.



3.7.6 State Encoding

When working with explicit state machines, we must address the issue of state encoding: what bit-vector value to associate with each state?

With implicit state machines, we do not need to worry about state encoding. The synthesis program determines the number of states and the encoding for each state.

3.7.6.1 Constants vs Enumerated Type

Using an enumerated type, the synthesis tools chooses the encoding:

```
type state_ty is (s0, s1, s2, s3);
signal state : state_ty;
```

Using constants, we choose the encoding:

```
type state_ty is std_logic_vector(1 downto 0);
constant s0 : state_ty := "11";
constant s1 : state_ty := "10";
constant s2 : state_ty := "00";
constant s3 : state_ty := "01";
signal state : state_ty;
```

Providing Encodings for Enumerated Types

Many synthesizers allow the user to provide hints on how to encode the states, or allow the user to provide explicitly the desired encoding. These hints are done either through VHDL attributes or special comments in the code.

Simulation

When doing functional simulation with enumerated types, simulators often display waveforms with “pretty-printed” values rather than bits (e.g. `s0` and `s1` rather than `11` and `10`). However, when simulating a design that has been mapped to gates, the enumerated type disappears and you are left with just bits. If you don’t know the encoding that the synthesis tool chose, it can be very difficult to debug the design.

However, this opens you up to potential bugs if the enumerated type you are testing grows to include more values, which then end up unintentionally executing your `when other` branch, rather than having a special branch of their own in the case statement.

Unused Values

If the number of values you have in your datatype is not a power of two, then you will have some unused values that are representable.

For example:

```
type state_ty is std_logic_vector(2 downto 0);
constant s0 : state_ty := "011";
constant s1 : state_ty := "000";
constant s2 : state_ty := "001";
constant s3 : state_ty := "011";
constant s4 : state_ty := "101";
signal state : state_ty;
```

This type only needs five unique values, but can represent eight different values. What should we do with the three representable values that we don’t need? The safest thing to do is to code your design so that if an illegal value is encountered, the machine resets or enters an error state.

3.7.6.2 Encoding Schemes

- Binary: Conventional binary counter.
- One-hot: Exactly one bit is asserted at any time.
- Modified one-hot: Altera’s Quartus synthesizer generates an almost-one-hot encoding where the bit representing the reset state is inverted. This means that the reset state is all ‘0’s and all other states have two ‘1’s: one for the reset state and one for the current state.
- Gray: Transition between adjacent values requires exactly one bit flip.
- Custom: Choose encoding to simplify combinational logic for specific task.

Tradeoffs in Encoding Schemes

- Gray is good for low-power applications where consecutive data values typically differ by 1 (e.g. no random jumps).
- One-hot usually has less combinational logic and runs faster than binary for machines with up to a dozen or so states. With more than a dozen states, the extra flip-flops required by one-hot encoding become too expensive.
- Custom is great if you have lots of time and are incredibly intelligent, or have deep insight into the guts of your design.

NOTE: Don't care values *When we don't care what is the value of a signal we assign the signal ' - ', which is "don't care" in VHDL. This should allow the synthesis tool to use whatever value is most helpful in simplifying the Boolean equations for the signal (e.g. Karnaugh maps). In the past, some groups in E&CE 427 have used ' - ' quite successfully to decrease the area of their design. However, a few groups found that using ' - ' **increased** the size of their design, when they were expecting it to decrease the size. So, if you are tweaking your design to squeeze out the last few unneeded FPGA cells, pay close attention as to whether using ' - ' hurts or helps.*

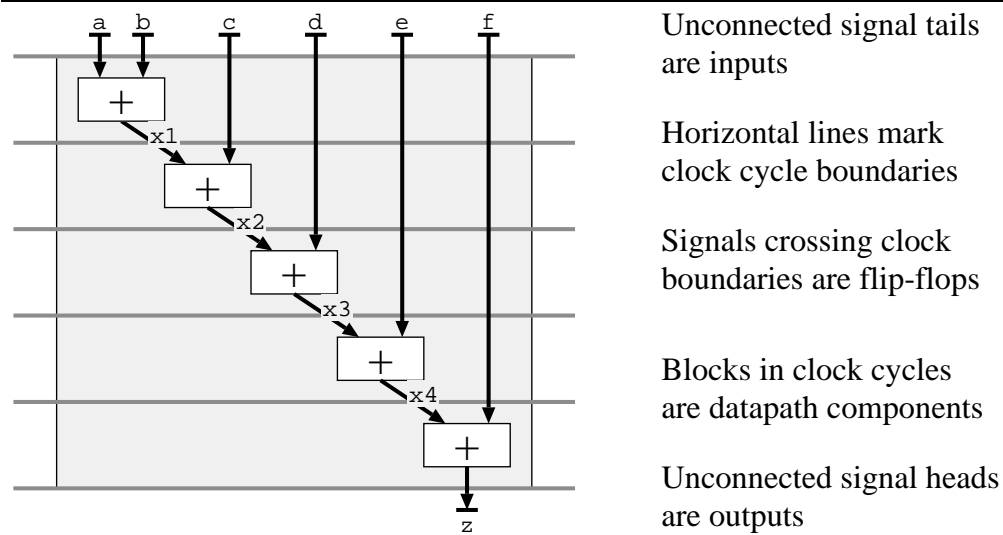
3.8 Dataflow Diagrams

3.8.1 Introduction to Dataflow Diagrams

3.8.1.1 Dataflow Diagrams Overview

- Dataflow diagrams are data-dependency graphs where the computation is divided into clock cycles.
- Purpose:
 - Provide a disciplined approach for designing datapath-centric circuits
 - Guide the design from algorithm, through high-level models, and finally to register transfer level code for the datapath and control circuitry.
 - Estimate area and performance
 - Make tradeoffs between different design options
- Background
 - Based on techniques from high-level synthesis tools
 - Some similarity between high-level synthesis and software compilation

Supplemental material available online (dataflow)



The use of memory arrays in dataflow diagrams is described in section 3.10.2.

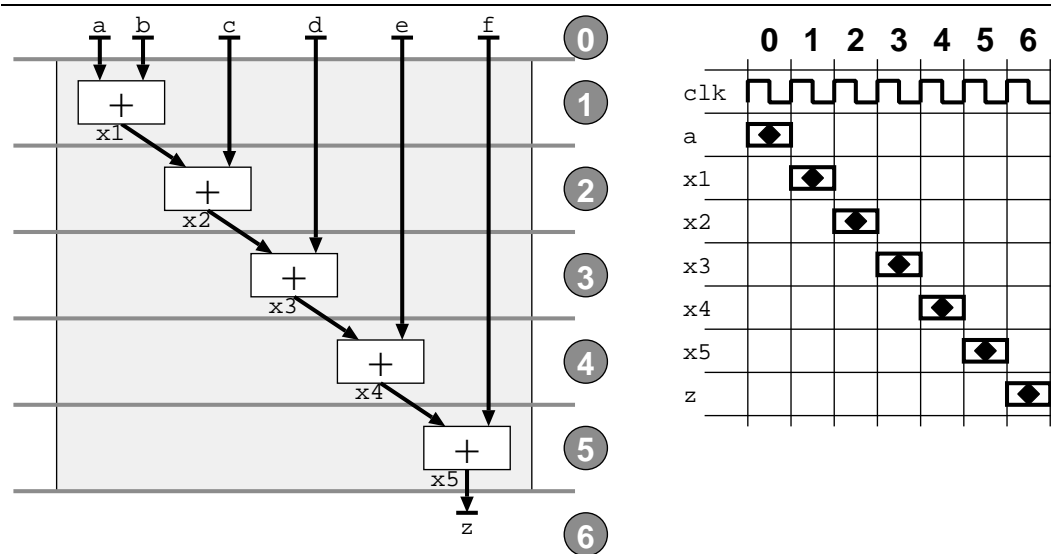
3.8.1.2 Area Estimation

- Maximum number of **blocks in a clock cycle** is total number of that **component** that are needed
- Maximum number of **signals that cross a cycle boundary** is total number of **registers** that are needed
- Maximum number of **unconnected signal tails in a clock cycle** is total number of **inputs** that are needed
- Maximum number of **unconnected signal heads in a clock cycle** is total number of **outputs** that are needed

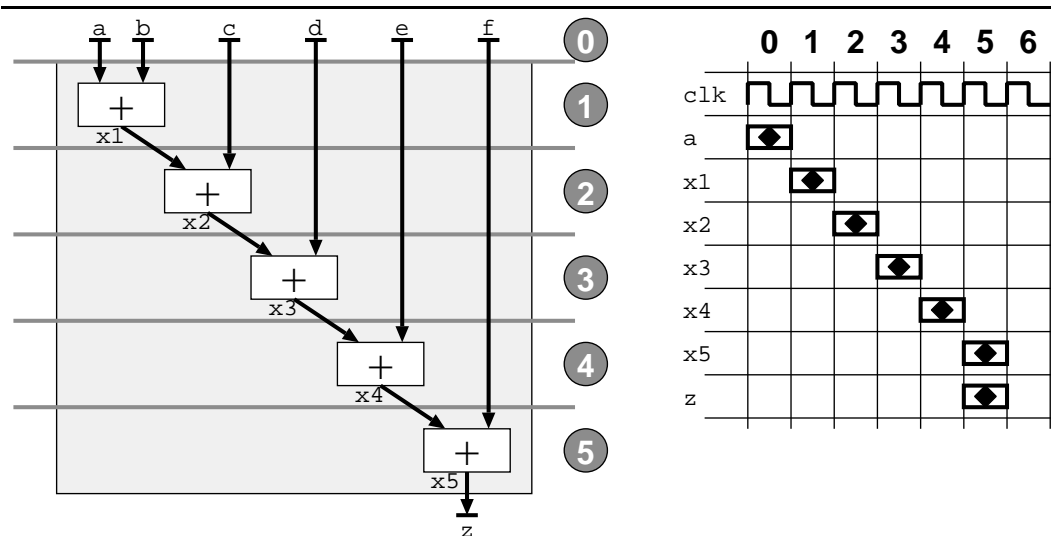
3.8.1.3 Dataflow Diagram Execution

Execution with Registers on Both Inputs and Outputs

Supplemental material available online (exec-reg)



Execution Without Output Registers



3.8.1.4 Performance Estimation

Performance Equations

$$\text{Performance} \propto \frac{1}{\text{TimeExec}}$$

$$\text{TimeExec} = \text{Latency} \times \text{ClockPeriod}$$

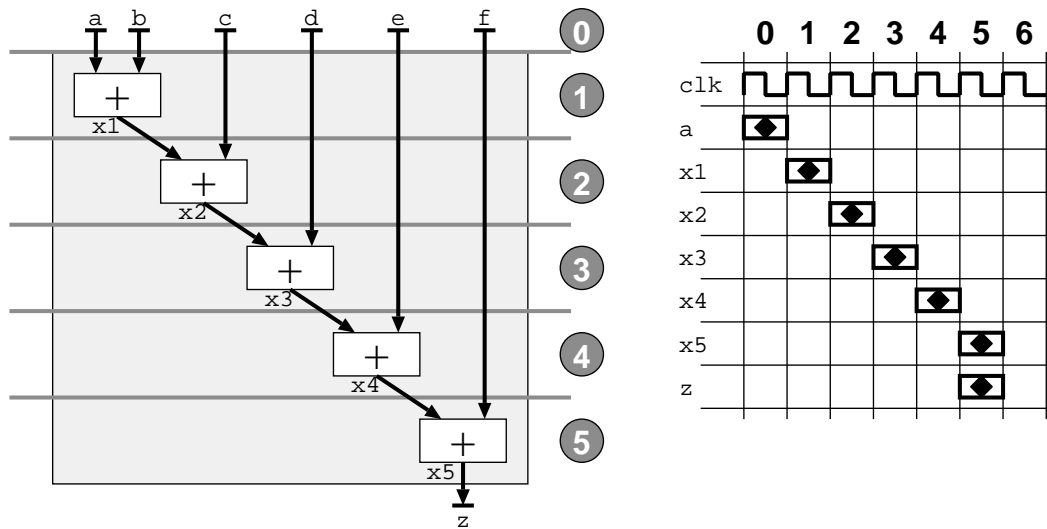
Latency = Number of clock cycles from inputs to outputs

There is much more information on performance in chapter ??, which is devoted to performance.

Performance of Dataflow Diagrams

- Latency: count horizontal lines in diagram
- Min clock period (Max clock speed) limited by longest path in a clock cycle

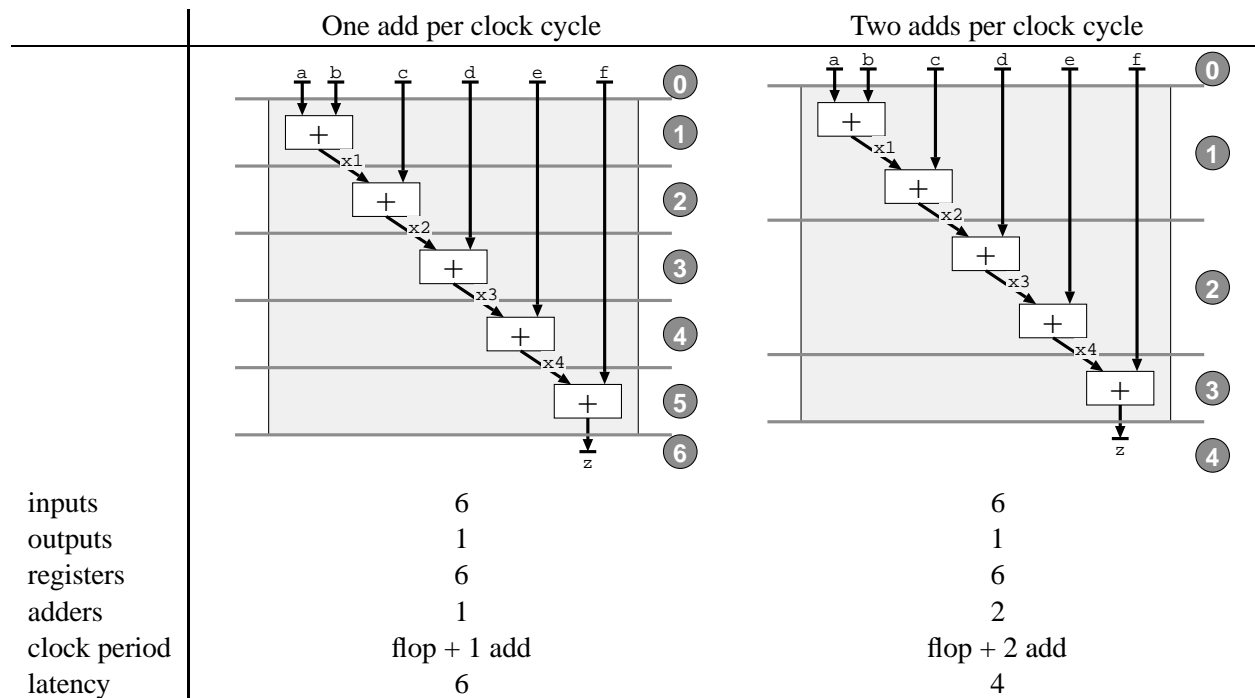
3.8.1.5 Design Analysis



num inputs	6
num outputs	1
num registers	6
num adders	1
min clock period	delay through flop and one adder
latency	5 clock cycles

3.8.1.6 Area / Performance Tradeoffs

Supplemental material available online (area-perf)



Question: Under what circumstances would each of the design options (one add and two add) be the fastest?

3.9 Datapath-Centric Design Example

We'll go through the following artifacts:

1. requirements
2. algorithm
3. high-level models
4. dataflow diagram
5. hardware block diagram
6. RTL code for datapath
7. state machine
8. RTL code for control

3.9.1 Requirements

Functional requirements:

- Compute the sum of six 8-bit numbers: $\text{output} = a + b + c + d + e + f$
- Use registers on both inputs and outputs

Performance requirements:

- Maximum clock period: unlimited
- Maximum latency: four

Cost requirements:

- Maximum of two adders
- Small miscellaneous hardware (e.g. muxes) is unlimited
- Maximum of three inputs and one output
- Design effort is unlimited

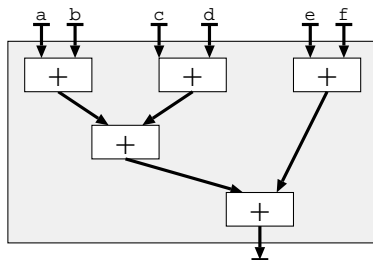
NB: In reality multiplexers are not free. In FPGAs, a 2:1 mux is more expensive than a full-adder. A 2:1 mux has three inputs while an adder has only two inputs (the carry-in and carry-out signals usually use the special “vertical” connections on the FPGA cell). In FPGAs, sharing an adder between two signals can be more expensive than having two adders. In a “generic-gate” technology, a multiplexor contains three two-input gates, while a full-adder contains fourteen two-input gates.

3.9.2 Algorithm

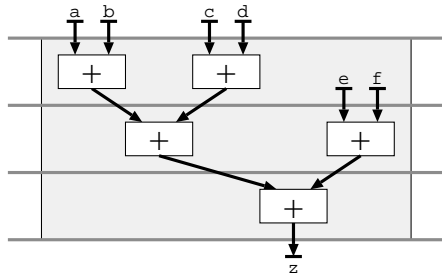
We’ll use parentheses to group operations so as to maximize our opportunities to perform the work in parallel:

$$z = (a + b) + (c + d) + (e + f)$$

This results in the following data-dependency graph:



3.9.3 Initial Dataflow Diagram



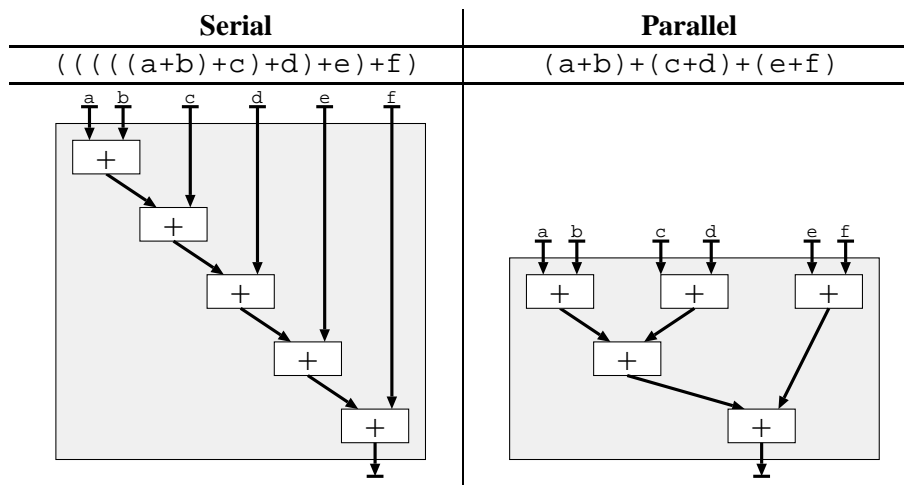
This dataflow diagram violates the requirement to use at most three inputs.

3.9.4 Dataflow Diagram Scheduling

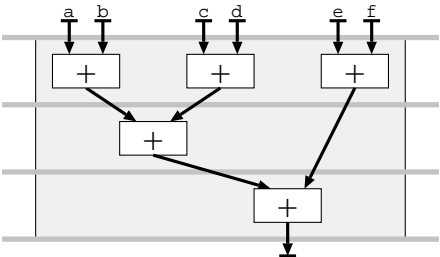
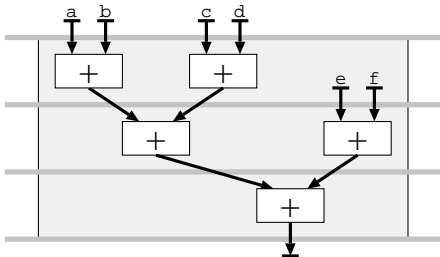
We can potentially optimize the inputs, outputs, area, and performance of a dataflow diagram by rescheduling the operations, that is allocating the operations to different clock cycles.

Parallel algorithms have higher performance and greater scheduling flexibility than serial algorithms

Serial algorithms tend to have less area than parallel algorithms



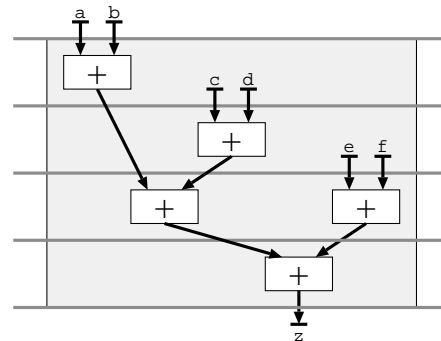
Scheduling to Optimize Area

	Original parallel	Parallel after scheduling
		
inputs	6	4
outputs	1	1
registers	6	4
adders	3	2
clock period	flop + 1 add	flop + 1 add
latency	3	3

Scheduling to Optimize Inputs

Rescheduling the dataflow diagram from the parallel algorithm reduced the area from three adders to two. However, it still violates the restriction of a maximum of three inputs. We can reschedule the operations to keep the same area, but reduce the number of inputs.

The tradeoff is that reducing the number of inputs causes an increase in the latency from four to five.



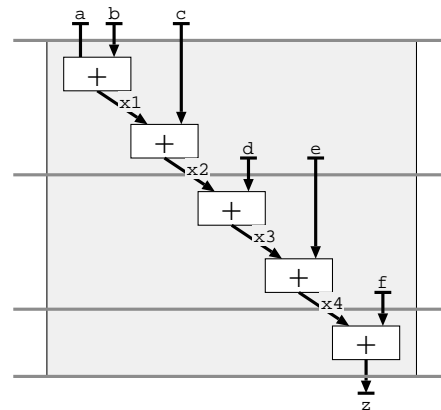
A latency of five violates the design requirement of a maximum latency of five clock cycles. In comparing the dataflow diagram above with the design requirements, we notice that the requirements allow a clock cycle that includes two additions and three inputs.

It appears that the parallel algorithm will not lead us to a design that satisfies the requirements.

We revisit the algorithm and try a serial algorithm:

$$z = (((a + b) + c) + d) + e + f$$

The corresponding dataflow diagram is shown to the right.

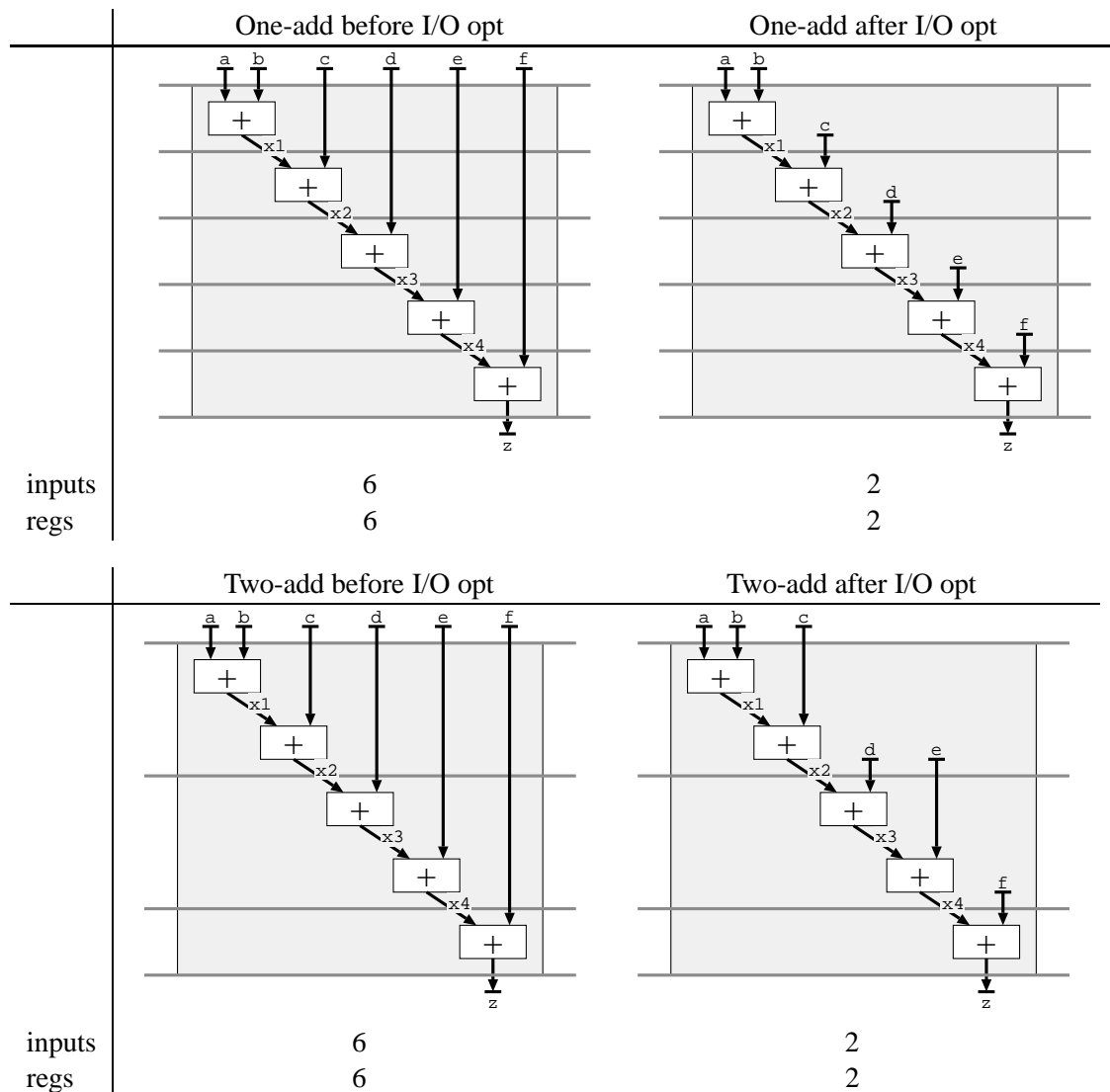


3.9.5 Optimize Inputs and Outputs

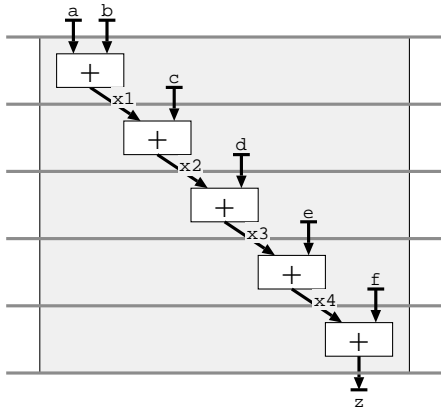
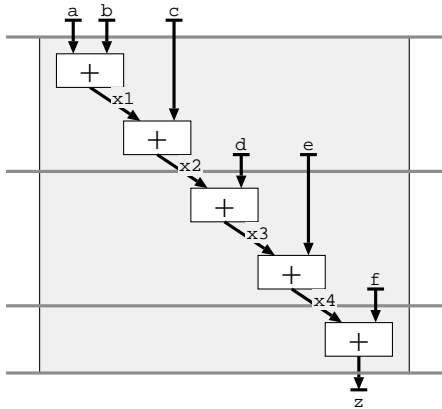
When we rescheduled the parallel algorithm, we rescheduled the input values. This requires renegotiating the schedule of input values with our environment. Sometimes the environment of our circuit will be willing to reschedule the inputs, but in other situations the environment will impose a non-negotiable schedule upon us.

If you are currently storing all inputs and can change environment's behaviour to delay sending some inputs, then you can reduce the number of inputs and registers.

We will illustrate this on both the one-add and the two-add designs.

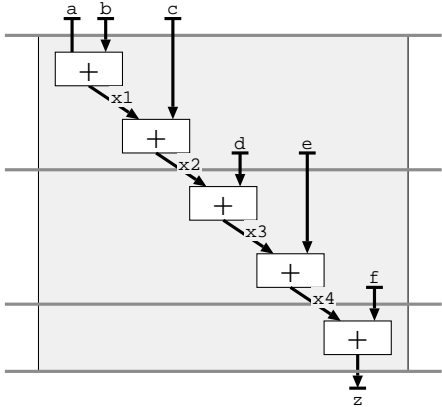


Design Comparison Between One and Two Add

	One-add after I/O opt	Two-add after I/O opt
		
inputs	2	3
outputs	1	1
registers	2	3
adders	1	2
clock period	flop + 1 add	flop + 2 add
latency	6	4

Hardware Recipe for Two-Add

We return now to the two-add design, with the dataflow diagram:



Based on the dataflow diagram, we can determine the hardware resources required for the datapath.

Table 3.2: Hardware Recipe for Two-Add

inputs	3
adders	2
registers	3
output	1
registered inputs	YES
registered outputs	YES
clock cycles from inputs to outputs	4

3.9.6 Input/Output Allocation

Our first step after settling on a hardware recipe is I/O allocation, because that determines the interface between our circuit and the outside world.

From the hardware recipe, we know that we need only three inputs and one output. However, we have six different input values. We need to allocate these input values to input signals before we can write a high-level model that performs the computation of our design.

Based on the input and output information in the hardware recipe, we can define our entity:

```
entity big_add is
  port (
    i1, i2, i3 : in unsigned(7 downto 0);
    o1          : out unsigned(7 downto 0)
  );
end big_add;
```

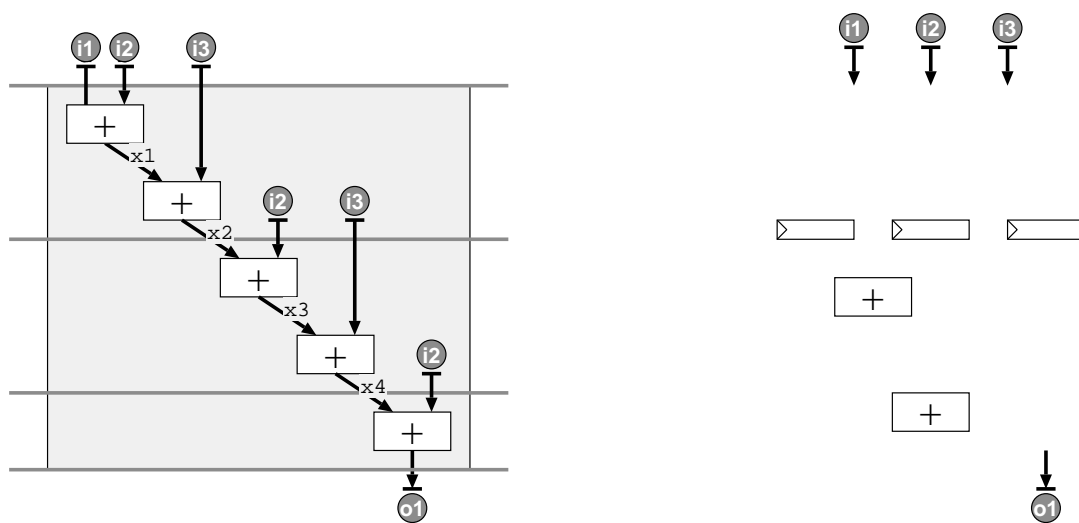


Figure 3.5: Dataflow diagram and hardware block diagram with I/O port allocation

Based upon the dataflow diagram after I/O allocation, we can write our first high-level model (hlm_v1).

In the high-level model the entire circuit will be implemented in a single process. For larger circuits it may be beneficial to have separate processes for different groups of signals.

In the high-level model, the code between wait statements describes the work that is done in a clock cycle.

The hlm architecture uses an implicit state machine.

Because the process is clocked, all of the signals that are assigned to in the process are registers. Combinational signals would need to be done using concurrent assignments or combinational processes.

```
architecture hlm_v1 of big_add is
  ...internal signal decls...
  process begin
    wait until rising_edge(clk);
    a <= i1;
    b <= i2;
    c <= i3;
    wait until rising_edge(clk);
    x2 <= (a + b) + c;
    d <= i2;
    e <= i3;
    wait until rising_edge(clk);
    x4 <= (x2 + d) + e;
    f <= i2;
    wait until rising_edge(clk);
    z <= (x4 + f);
  end process;
  o1 <= z;
end hlm_v1;
```

3.9.7 Register Allocation

The next step after I/O allocation could be either register allocation or datapath allocation. The benefit of doing register allocation first is that it is possible to write VHDL code after register allocation is done but before datapath allocation is done, while the inverse (datapath done but register allocation not done) does not make sense if written in a hardware description language. In this example, we will do register allocation before datapath allocation, and show the resulting VHDL code.

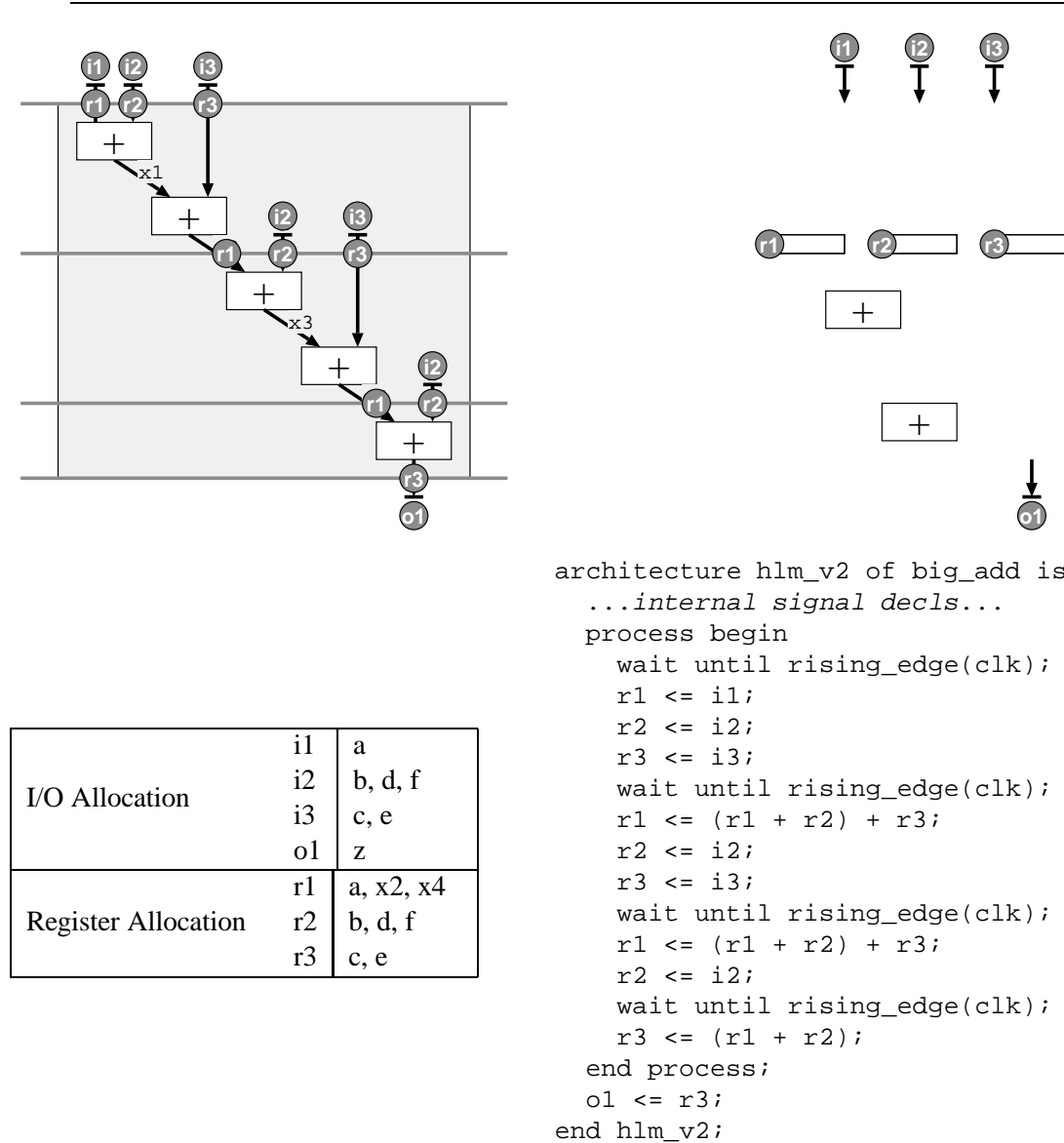
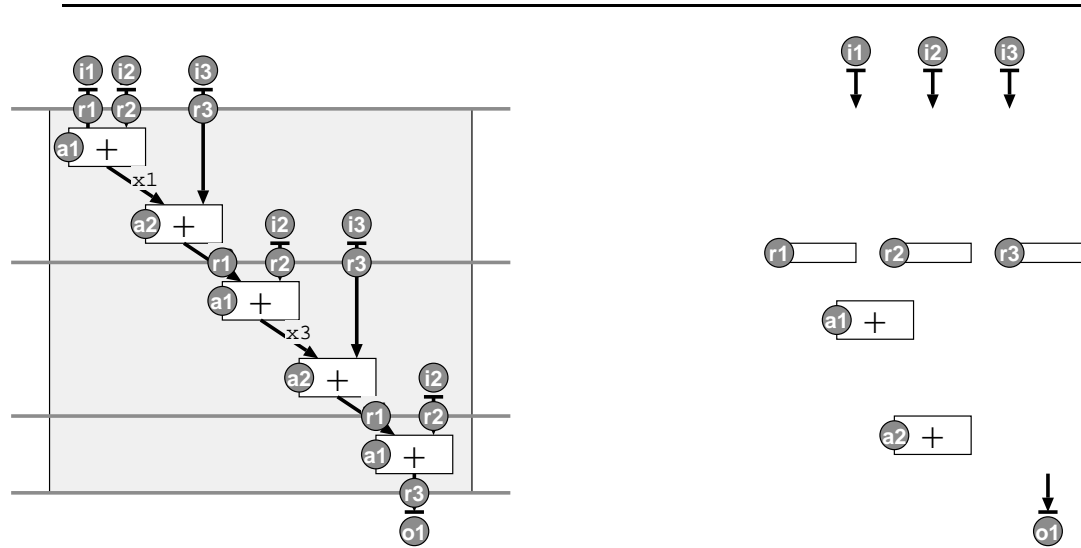


Figure 3.6: Block diagram after I/O and register allocation

3.9.8 Datapath Allocation

In datapath allocation, we allocate each of the data operations in the dataflow diagram to one of the datapath components in the hardware block diagram.



I/O Allocation	i1	a
	i2	b, d, f
	i3	c, e
	o1	z
Register Allocation	r1	a, x2, x4
	r2	b, d, f
	r3	c, e
Datapath Allocation	a1	x1, x3, z
	a2	x2, x4

```

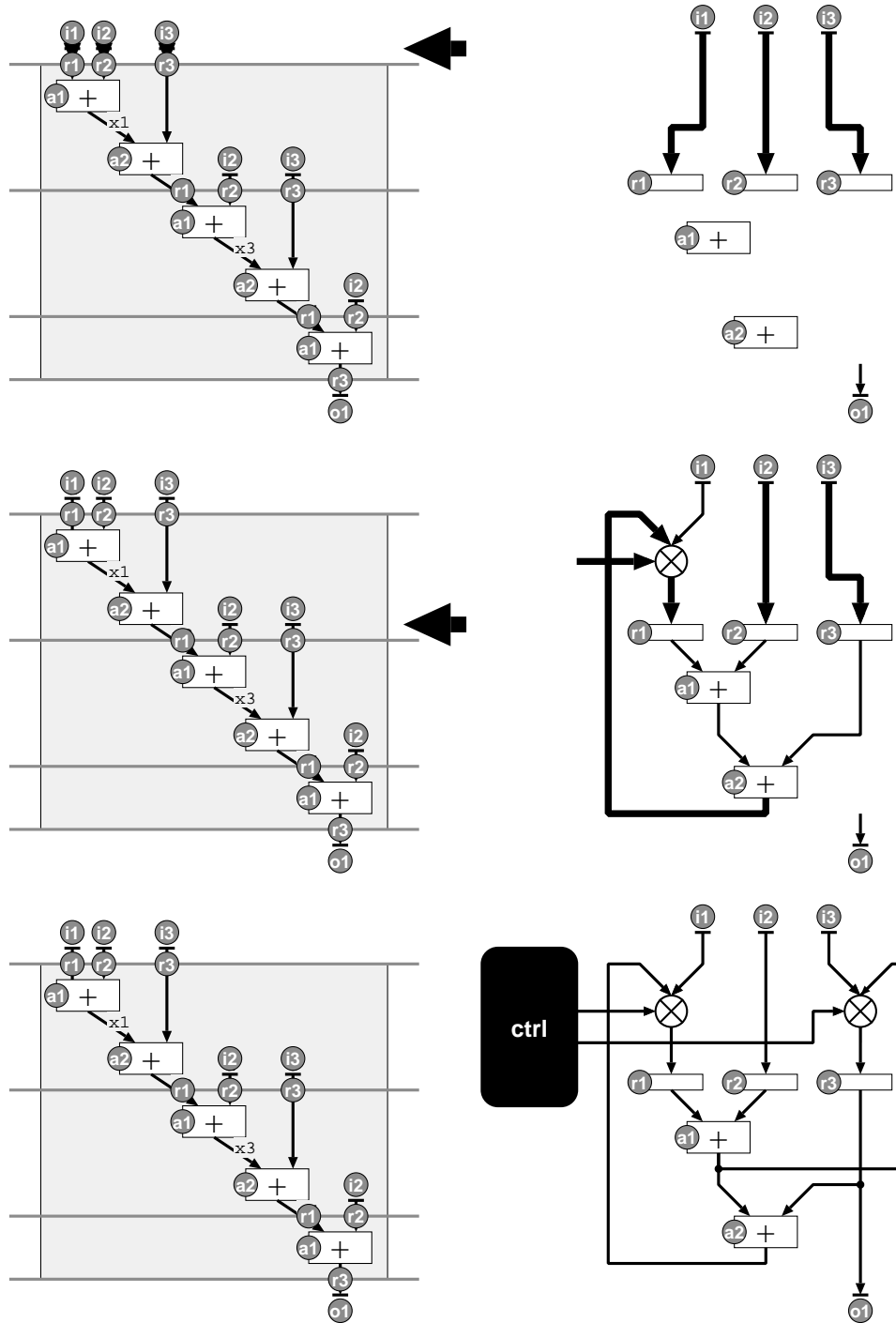
architecture hlm_dp of big_add is
  ...internal signal decls...
  process begin
    wait until rising_edge(clk);
    r1 <= i1;
    r2 <= i2;
    r3 <= i3;
    wait until rising_edge(clk);
    r1 <= a2;
    r2 <= i2;
    r3 <= i3;
    wait until rising_edge(clk);
    r1 <= a2;
    r2 <= i2;
    wait until rising_edge(clk);
    r3 <= a1;
  end process;
  a1 <= r1 + r2;
  a2 <= a1 + r3;
  o1 <= r3;
end hlm_dp;

```

Figure 3.7: Block diagram after I/O, register, and datapath allocation

3.9.9 Datapath for DP+Ctrl Model

We execute/simulate the dataflow diagram to calculate the connections between the datapath components and add multiplexers where needed.



Datapath

The following VHDL code is derived directly from the block diagram.

```
architecture main of big_add is
  fsm : process ... end;
  process (clk) begin
    if rising_edge(clk) then
      if r1_gets_in = '1' then
        r1 <= i1;
      else
        r1 <= a2;
      end if;
    end if;
  end process;
  process (clk) begin
    if rising_edge(clk) then
      r2 <= i2;
    end if;
  end process;
  process (clk) begin
    if rising_edge(clk) then
      if r3_gets_in = '1' then
        r3 <= i3;
      else
        r3 <= a1;
      end if;
    end if;
  end process;
  a1 <= r1 + r2;
  a2 <= a1 + r3;
  o1 <= r3;
end main;
```

3.9.10 Add State Machine

The state machine keeps track of which clock cycle of the dataflow diagram is currently being executed.

The state machine drives the datapath signals whose values are dependent upon which clock cycle of the dataflow diagram is being executed. Typical examples of these signals are:

- Select signals on multiplexers
- Instruction signals on arithmetic modules
- Chip-enable lines on registers and flip-flops

3.9.11 Summary: From Dataflow to Datapath Hardware

1. Create dataflow diagram
2. Optimize inputs and outputs
3. Schedule data operations
4. I/O allocation: assign dataflow signals to hardware inputs and outputs
5. Register allocation: assign dataflow signals to registers
6. Datapath allocation: assign dataflow blocks to components
7. Connect the datapath blocks, add muxes where needed
8. Derive datapath for DP+Ctrl model
9. Build the state machine, connect to datapath + storage

3.9.12 From Dataflow to State Machine

Two control signals from state machine:

`r1_gets_in` `r1` reads from input or `a2`
`r3_gets_in` `r3` reads from input or `a1`

Simulate dataflow diagram and record required values of signals.

cycle	<code>r1_gets_in</code>	<code>r3_gets_in</code>
1	true	true
2	false	true
3	false	—
4	—	false

The control signals for the datapath (`r1_gets_in` and `r3_gets_in`) drive the two multiplexors, one for each register (`r1` and `r3`).

The values of `r1_gets_in` and `r3_gets_in` are determined solely by the current state of the machine.

3.9.13 Implicit State Machine

Because dataflow diagrams tend to have simple control flows, the implicit style of writing state machines is generally well suited.

```
process (clk) begin
    -----
    -- cycle 1
    wait until rising_edge(clk);
    r1_gets_in <= '1';
    r3_gets_in <= '1';
    -----
    -- cycle 2
    wait until rising_edge(clk);
    r1_gets_in <= '0';
    r3_gets_in <= '1';
    -----
    -- cycle 3
    wait until rising_edge(clk);
    r1_gets_in <= '0';
    r3_gets_in <= '-';
    -----
    -- cycle 4
    wait until rising_edge(clk);
    r1_gets_in <= '-';
    r3_gets_in <= '0';
end process;
```

In cycle 3, we don't care what is the value of `r3_gets_in`. In cycle 4, we don't care what the value of `r1_gets_in` is. So we assign these signals `'-'` in these clock cycles, which is “don't care” in VHDL.

3.9.14 Explicit-Current+Next State Machines

A clocked process is used to store the state and a concurrent assignment is used to calculate the next state.

We write the clocked process for the current state, and then look at several different coding styles for calculating the next state.

3.9.14.1 Current-State Process

```

architecture main of big_add is
    type state_ty is (S0, S1, S2, S3);
    signal state, state_nxt : state_ty;
    ...
begin
    process (clk) begin
        if rising_edge(clk) then
            state_cur <= state_nxt;
        end if;
    end process;
    with state_cur select
        state_nxt <= S1 when S0,
                    S2 when S1,
                    S3 when S2,
                    S0 when S3
    ;
    ...r1_gets_in asn...
    ...r3_gets_in asn...
    ...datapath...
end main;

```

3.9.14.2 Next-State: Conditional Assignment

The first coding example uses simple conditional assignments.

```

r1_gets_in <= '1' when state_cur = S0
              else '0';
r3_gets_in <= '1' when state_cur = S3
              else '0';

```

3.9.14.3 Next-State: Conditional Assignment with Don't Care

The simple conditional assignment doesn't take advantage of the fact that the last state doesn't use the adder a1, so we don't care whether r1 reads from the input or from the a2.

We give the synthesis tool a chance to simplify equations for r1_gets_in (and thereby hopefully reduce area) by putting a don't care value for r1_gets_in in the last state.

```

r1_gets_in <= '1' when state_cur = S0
              else '0' when (state_cur = S1)
                           OR (state_cur = S2)
              else '-';
r3_gets_in <= '1' when (state_cur = S0)
               OR (state_cur = S1)
              else '0' when (state_cur = S4);
              else '-';

```

3.9.14.4 Next-State: Selected Assignment with Don't Care

The conditional assignment code has many occurrences of `state_cur` in the conditions, which is ugly. So, use a case-like statement (the selected assignment).

```
with state_cur select
    r1_gets_in <= '0' when S0,
                '1' when S1 | S2,
                '--' when others
;
with state_cur select
    r3_gets_in <= '0' when S3
                '1' when S0 | S1,
                '--' when others
;
```

3.9.14.5 Next-State: Case Statement

The selected assignment code tests `state_cur` for both assignments, so try a case statement in a process, which allows multiple assignments within the case statement.

```
process (state_cur)
begin
    case state_cur is
        when S0 =>
            r1_gets_in <= '1';
            r3_gets_in <= '1';
        when S1 =>
            r1_gets_in <= '0';
            r3_gets_in <= '1';
        when S2 =>
            r1_gets_in <= '0';
            r3_gets_in <= '--';
        when S3 =>
            r1_gets_in <= '--';
            r3_gets_in <= '0';
        end case;
    end process;
```

After writing out the different options, the selected assignment style looks to be the best option for this example. The code is short, clean and easy to understand.

3.10 Memory Arrays and RTL Design

3.10.1 Memory Arrays in VHDL

3.10.1.1 Using a Two-Dimensional Array for Memory

A memory array can be written in VHDL as a two-dimensional array:

```

subtype data is std_logic_vector(7 downto 0);
type data_vector is array( natural range <> ) of data;
signal mem : data_vector(31 downto 0);

```

However, a two-dimensional array does not accurately capture the limitations on a memory array in hardware.

The example below illustrates: lack of interface protocol, combinational write, multiple write ports, multiple read ports.

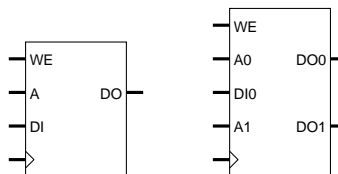
```

architecture main of mem_not_hw is
  subtype data is std_logic_vector(7 downto 0);
  type data_vector is array( natural range <> ) of data;
  signal mem : data_vector(31 downto 0);
begin
  y <= mem( a );
  mem( a ) <= b;           -- comb read
  process (clk) begin
    if rising_edge(clk) then
      mem( c ) <= w;       -- write port #1
    end if;
  end process;
  process (clk) begin
    if rising_edge(clk) then
      mem( d ) <= v;       -- write port #2
    end if;
  end process;
  u <= mem( e );           -- read port #2
end main;

```

3.10.1.2 Memory Arrays in Hardware

Most simple memory arrays are single- or dual-ported, support just one write operation at a time, and have an interface protocol using a clock and write-enable.



3.10.1.3 VHDL Code for Single-Port Memory Array

```

package mem_pkg is
  subtype data is std_logic_vector(7 downto 0);
  type data_vector is array( natural range <> ) of data;
end;

entity mem is
  port (
    clk  : in  std_logic;
    we   : in  std_logic           -- write enable
    a    : in  unsigned(4 downto 0); -- address
    di   : in  data;              -- data_in
    do   : out data                -- data_out
  );
end mem;
architecture main of mem is
  signal mem : data_vector(31 downto 0);
begin
  do <= mem( to_integer( a ) );
  process (clk) begin
    if rising_edge(clk) then
      if we = '1' then
        mem( to_integer( a ) ) <= di;
      end if;
    end if;
  end process;
end main;

```

Synopsys synthesis tools implement each bit in a two-dimensional array as a flip-flop.

Each FPGA and ASIC vendors supplies libraries of memory arrays that are smaller and faster than a two-dimensional array of flip flops. These libraries exploit specialized hardware on the chips to implement the memory.

NB: To synthesize a reasonable implementation of a memory array with Synopsys, you must instantiate a vendor-supplied memory component.

Some other synthesis tools can infer memory arrays from two-dimensional arrays and synthesize efficient implementations.

Recommended Design Process with Memory

1. high-level model with two-dimensional array
2. two-dimensional array packaged inside memory entity/architecture
3. vendor-supplied component

3.10.1.4 Using Library Components for Memory**Altera**

Altera uses “MegaFunctions” to implement RAM in VHDL. A MegaFunction is a black-box description of hardware on the FPGA. There are tools in Quartus to generate VHDL code for RAM components of different sizes. In E&CE 427 we will provide you with the VHDL code for the RAM components that you will need in Lab-3 and the Project.

The APEX20KE chips that we are using have dedicated SRAM blocks called *Embedded System Blocks* (ESB). Each ESB can store 2048 bits and can be configured in any of the following sizes:

Number of Elements	Word Size (bits)
2048	1
1024	2
512	4
256	8
128	16

Xilinx

Use component instantiation to get these components

ram16x1s 16 × 1 single ported memory

ram16x1d 16 × 1 dual-port memory

Other sizes are also available, consult the datasheet for your chip.

3.10.1.5 Build Memory from Slices

If the vendor's libraries of memory components do not include one that is the correct size for your needs, you can construct your own component from smaller ones.

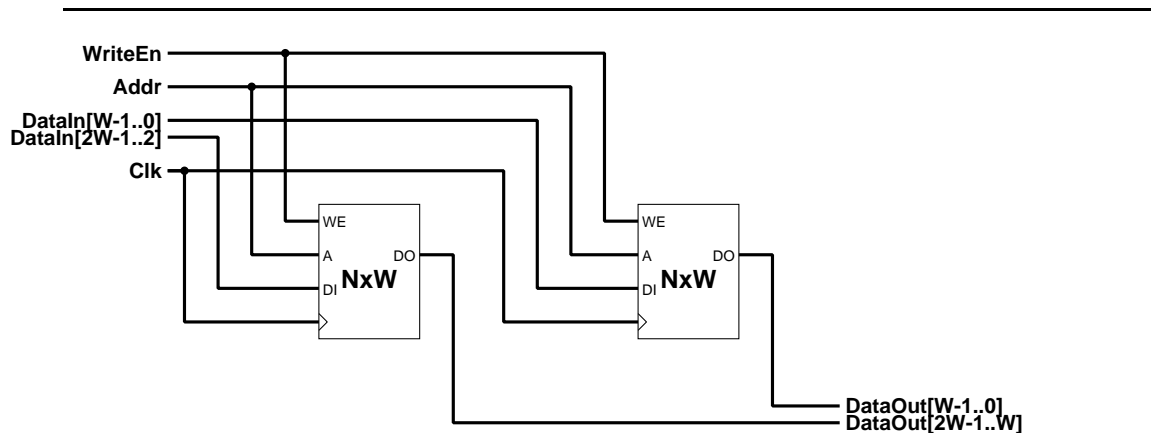


Figure 3.8: An $N \times 2W$ memory from $N \times W$ components

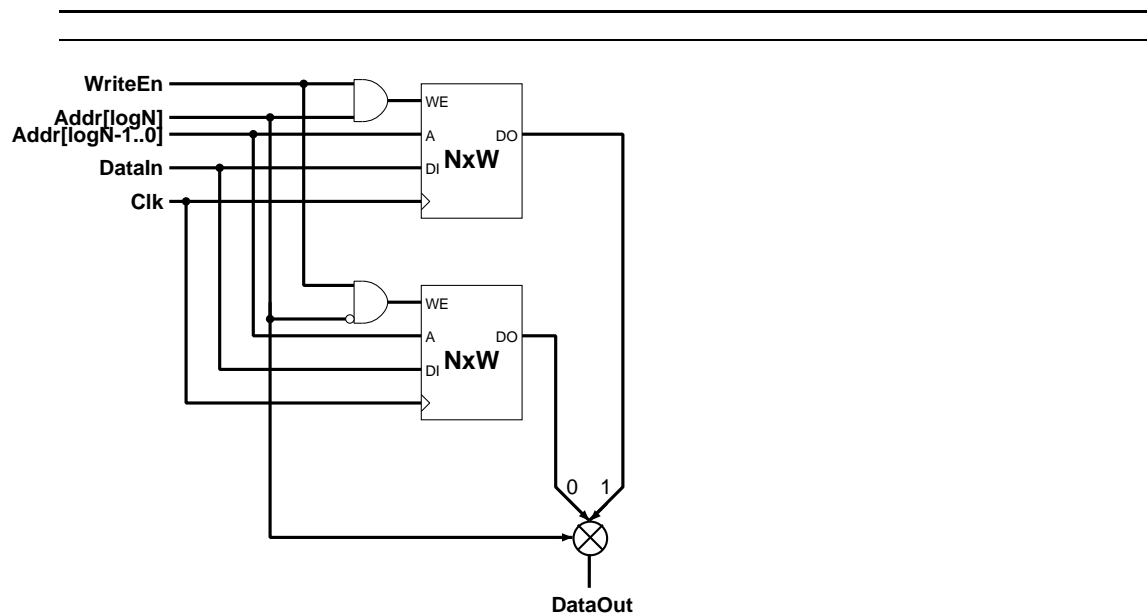


Figure 3.9: A $2N \times W$ memory from $N \times W$ components

A 16×4 Memory from 16×1 Components

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity raml6x4s is
  port (
    clk, we : in std_logic;
    data_in  : in  std_logic_vector(3 downto 0);
    addr     : in  unsigned(3 downto 0);
    data_out : out std_logic_vector(3 downto 0)
  );
end raml6x4s;

architecture main of raml6x4s is
  component raml6x1s
    port (d          : in std_logic;    -- data in
          a3, a2, a1, a0 : in std_logic; -- address
          we         : in std_logic;    -- write enable
          wclk       : in std_logic;    -- write clock
          o          : out std_logic    -- data out
    );
  end component;
begin
  mem_gen:
  for i in 0 to 3 generate
    ram : raml6x1s
      port map (
        we => we,
        wclk => clk,
        -----
        -- d and o are dependent on i
        a3 => addr(3),    a2 => addr(2),
        a1 => addr(1),    a0 => addr(0),
        d  => data_in(i),
        o  => data_out(i)
        -----
      );
    end generate;
end main;

```

3.10.1.6 Dual-Ported Memory

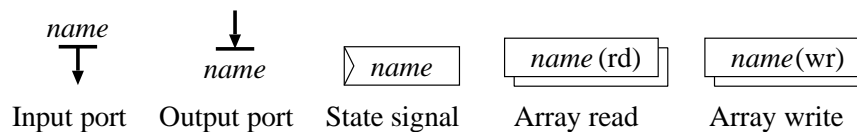
Dual ported memory is similar to single ported memory, except that it allows two simultaneous reads, or a simultaneous read and write.

When doing a simultaneous read and write to the same address, the read will **not** see the data currently being written.

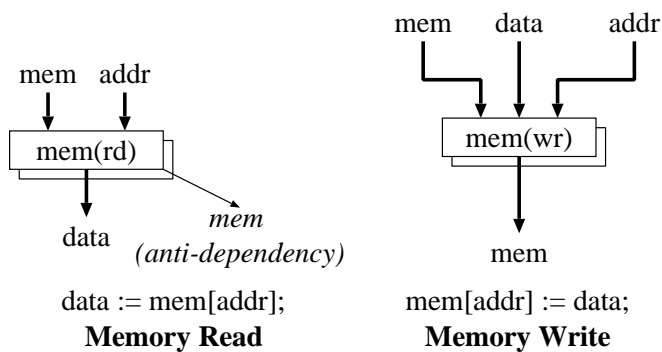
Question: *Why do dual-port memories usually not support writes on both ports?*

3.10.2 Memory Arrays and Dataflow Diagrams

3.10.2.1 Legend for Dataflow Diagrams



3.10.2.2 Basic Memory Operations



Dataflow diagrams show the dependencies between operations. The basic memory operations are similar, in that each arrow represents a data dependency.

There are a few aspects of the basic memory operations that are potentially surprising:

- The *anti-dependency* arrow producing *mem* on a read.
- Reads and writes are dependent upon the entire previous value of the memory array.
- The write operation appears to produce an entire memory array, rather than just updating an individual element of an existing array.

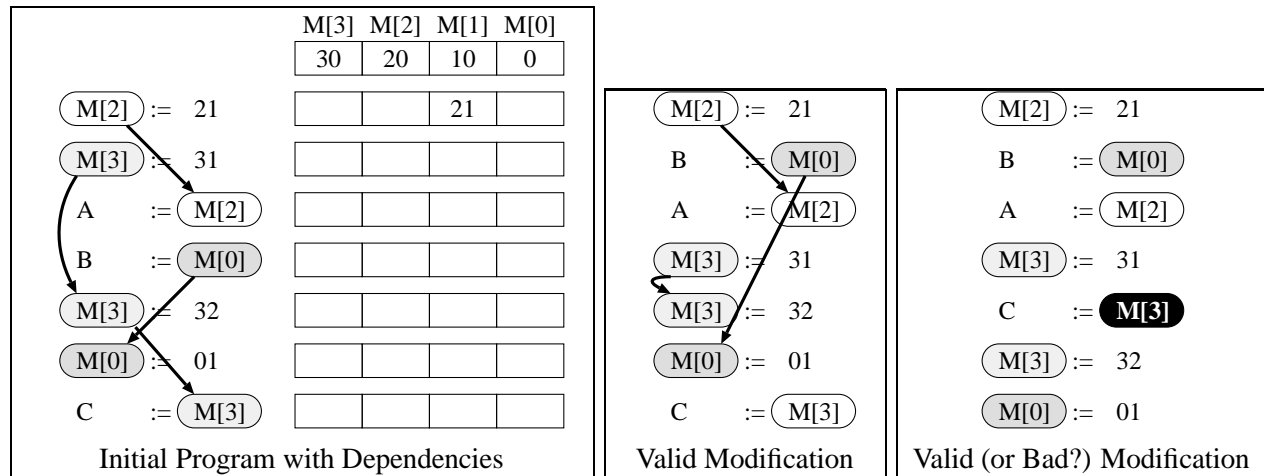
The antidependency for memory reads is related to Write-after-Read dependencies, as discussed in Section 3.10.2.4.

The apparent dependency on and production of an entire memory array is because we don't know which address in the array will be read from or written to. There are optimizations that can be performed when we know the address (Section 3.10.2.5).

3.10.2.3 Data Dependencies

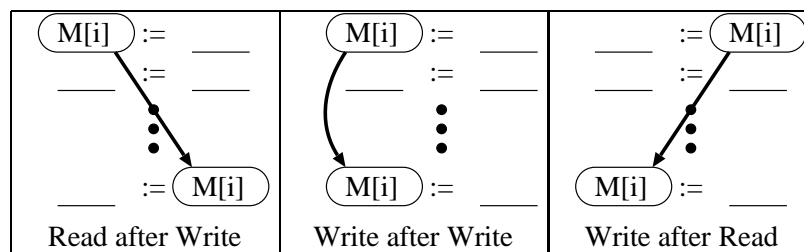
Instructions in a program can be reordered, so long as the data dependencies are preserved.

Supplemental material available online (data-dep)

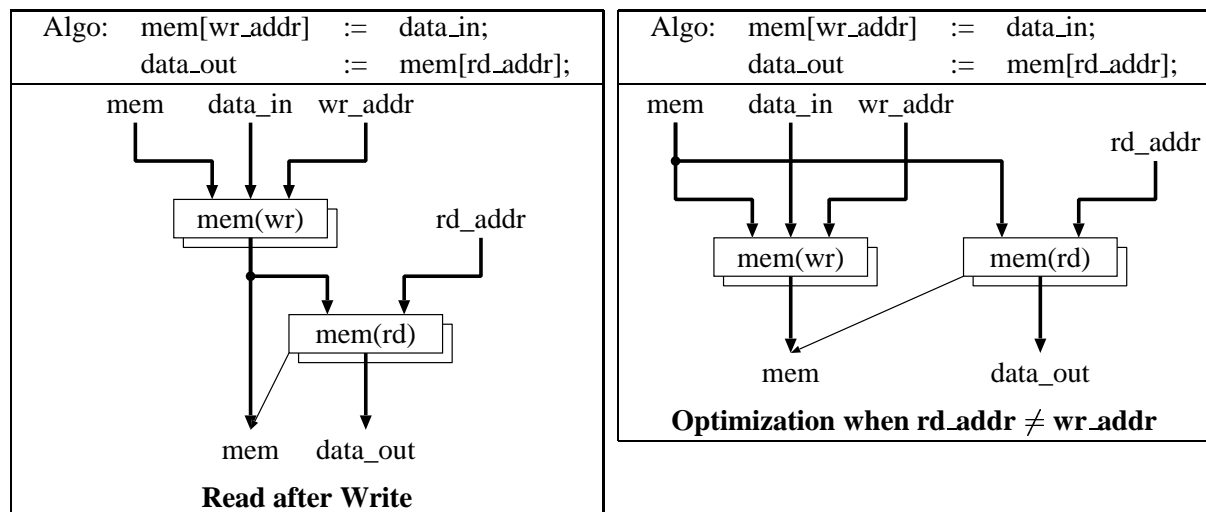


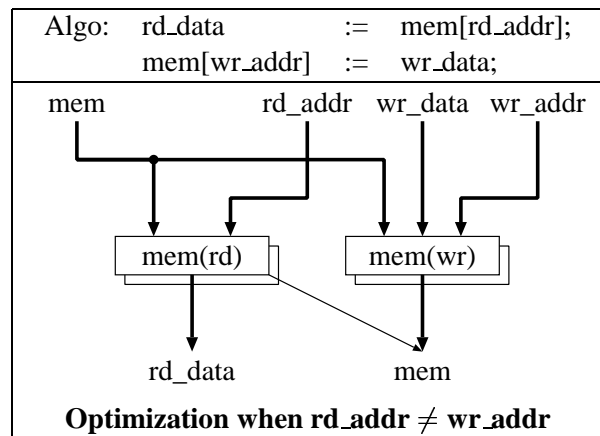
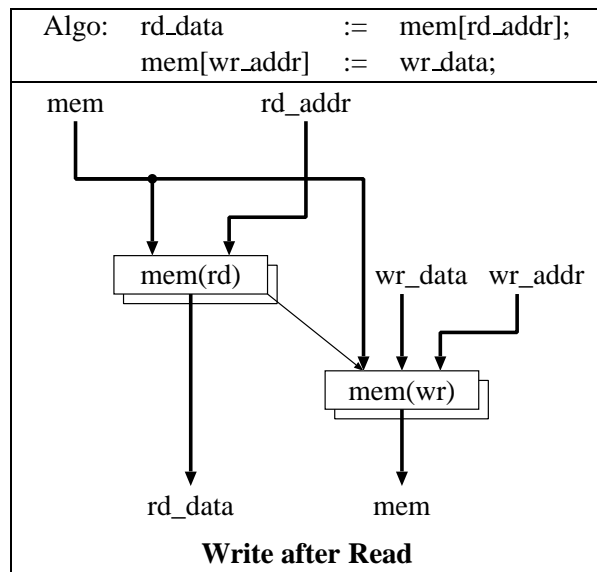
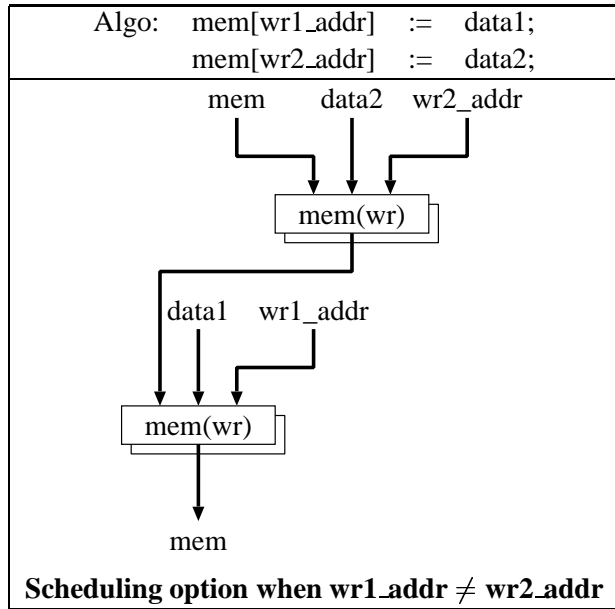
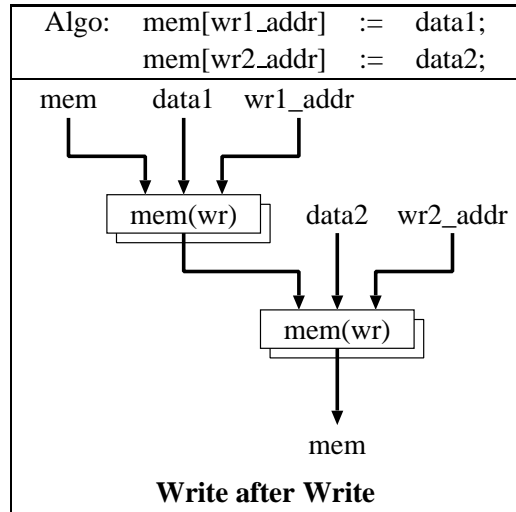
3.10.2.4 Definition of Three Types of Dependencies

There are three types of data dependencies. The names come from pipeline terminology in computer architecture.



3.10.2.5 Dataflow Diagrams and Data Dependencies





3.10.2.6 Example: Memory Array and Dataflow Diagram

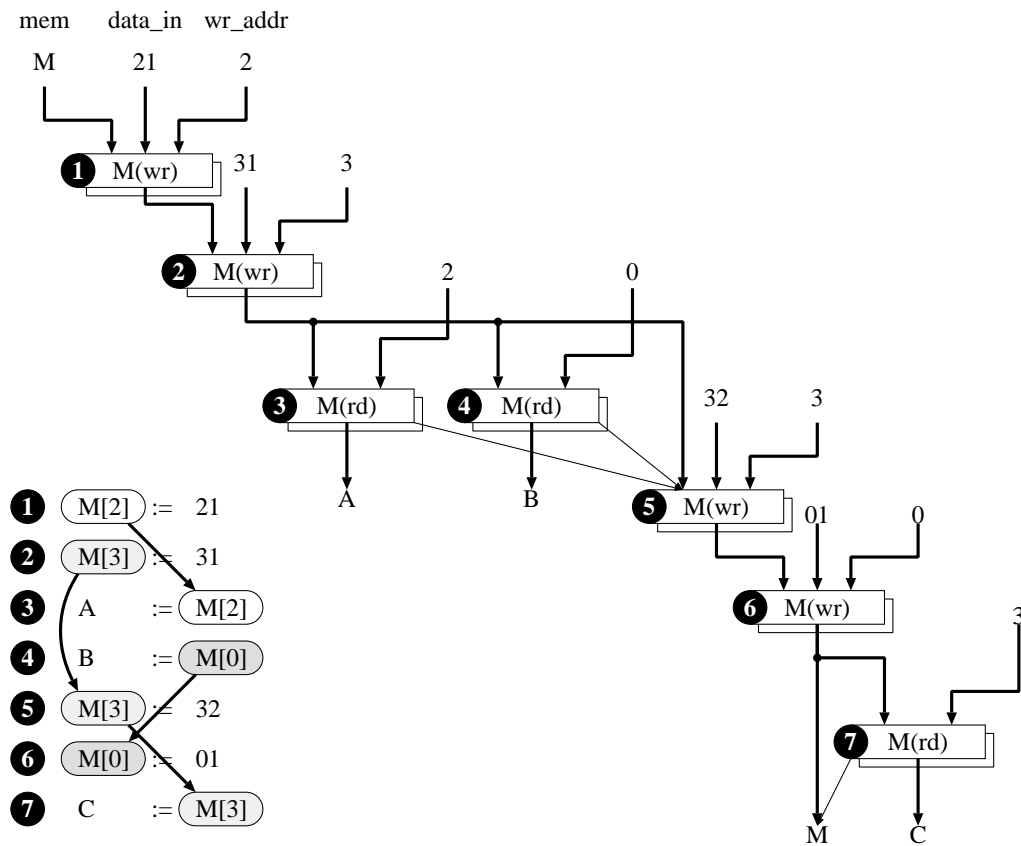


Figure 3.10: Memory array example code and initial dataflow diagram

The dependency and anti-dependency arrows in dataflow diagram in Figure 3.10 are based solely upon whether an operation is a read or a write. The arrows do not take into account the address that is read from or written to.

In figure 3.11, we have used knowledge about which addresses we are accessing to remove unneeded dependencies. These are the real dependencies and match those shown in the code fragment for figure 3.10. In figure 3.12 we have placed an ordering on the read operations and an ordering on the write operations. The ordering is derived by obeying data dependencies and then rearranging the operations to perform as many operations in parallel as possible.

Supplemental material available online (addr-opt)

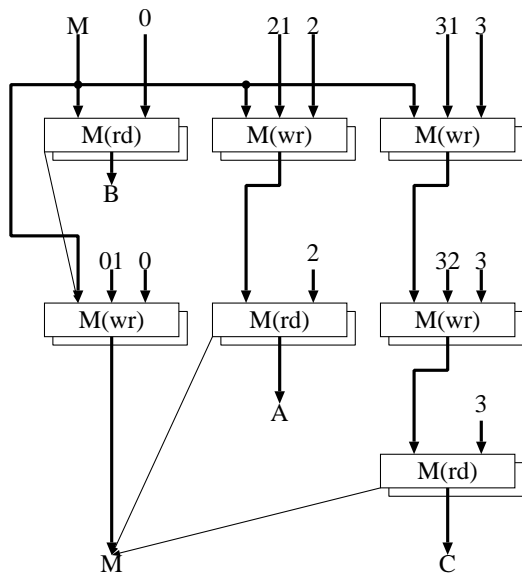


Figure 3.11: Memory array with minimal dependencies

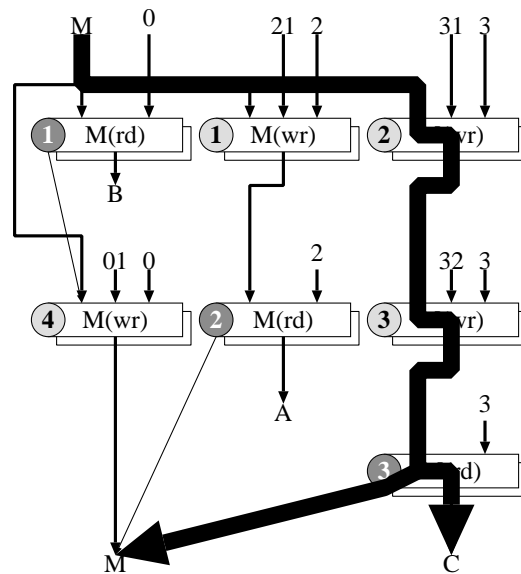


Figure 3.12: Memory array with orderings

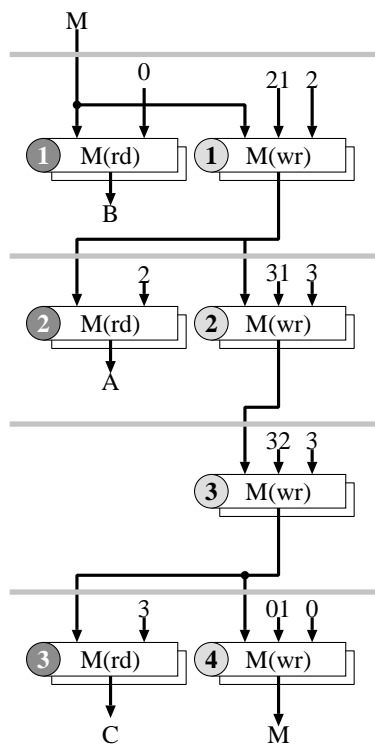


Figure 3.13: Final version of Figure 3.10

3.11 Input / Output Protocols

An important aspect of hardware design is choosing a input/output protocol that is easy to implement and suits both your circuit and your environment. Here are a few simple and common protocols.

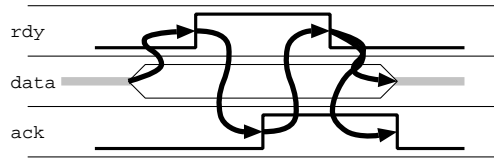


Figure 3.14: Four phase handshaking protocol

Used when timing of communication between producer and consumer is unpredictable. The disadvantage is that it is cumbersome to implement and slow to execute.

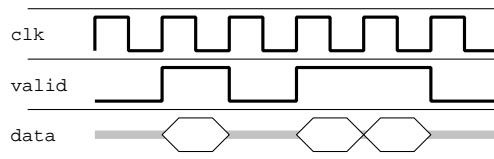


Figure 3.15: Valid-bit protocol

A low overhead (both in area and performance) protocol. Consumer must always be able to accept incoming data. Often used in pipelined circuits. More complicated versions of the protocol can handle pipeline stalls.

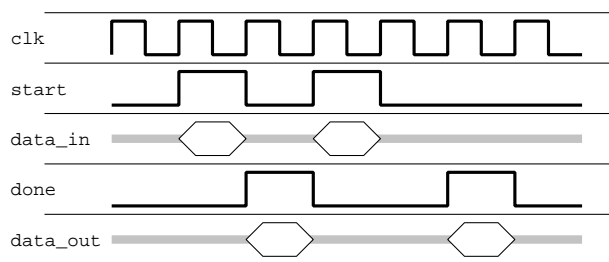


Figure 3.16: Start/Done protocol

A low overhead (both in area and performance) protocol. Useful when a circuit works on one piece of data at a time and the time to compute the result is unpredictable.

3.12 Design Example: Stack

3.12.1 Stack: Requirements

3.12.1.1 Entity

VHDL entity for the stack:

```
entity stack is
  port (
    reset, clk : in std_logic;
    inp        : in std_logic_vector(3 downto 0);
    outp       : out std_logic_vector(3 downto 0)
  );
end stack;
```

The input signal `inp` is used for both instructions and data.

3.12.1.2 Instructions

<code>push</code>	put a new piece of data onto the top of the stack
<code>pop</code>	remove the top piece of data from the stack
<code>swap</code>	swap the top two pieces of data
<code>tos</code>	output the current data on the top of the stack

3.12.1.3 Instruction Encoding

VHDL package defining stack instructions:

```
package stack_instr is
  constant pop   : std_logic_vector(3 downto 0) := "0001";
  constant push  : std_logic_vector(3 downto 0) := "0010";
  constant tos   : std_logic_vector(3 downto 0) := "0100";
  constant swap  : std_logic_vector(3 downto 0) := "1000";
end stack_instr;
```

3.12.1.4 Miscellaneous Requirements

- The stack shall have 16 elements
- The inputs shall be registered.
- When a push operation is done, in the clock cycle following the push instruction, `inp` shall have the data that is to be pushed onto the stack.
- Popping from an empty stack or pushing onto a full stack results in undefined behaviour.
- When doing a `tos` or `pop` operation, the output `outp` shall have the `tos` data in the clock cycle after the `tos` instruction is input. At all other times the output is unconstrained.
- In the clock cycle following `reset` being asserted (set to '1'), the stack shall be empty.

3.12.2 Stack: Algorithm

A simple Perl program to implement an algorithmic description of the stack.

NB: You don't need to know Perl in E&CE 427. Perl is just one example of the many different software programming languages that can be used to create algorithmic descriptions of circuits.

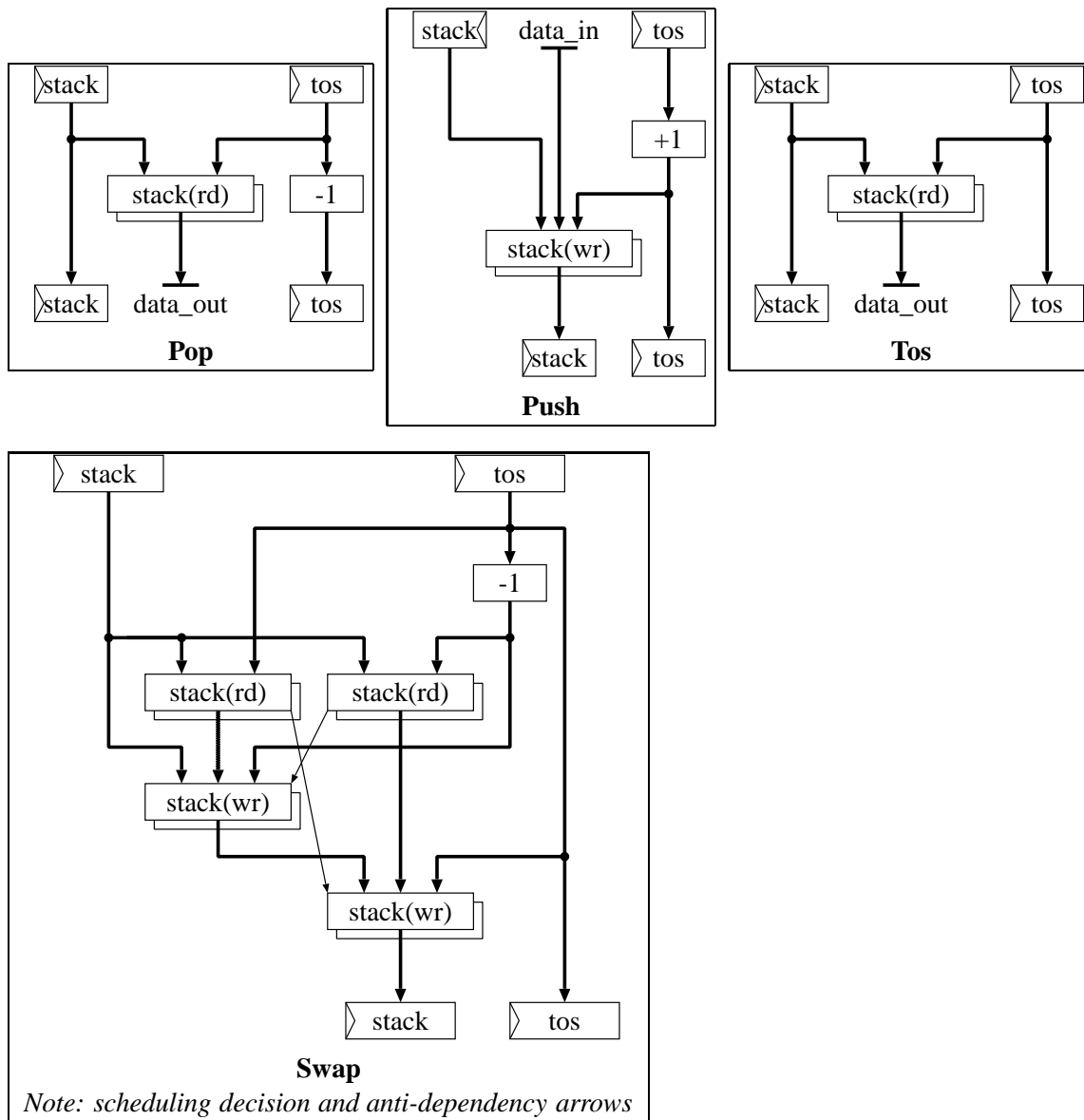
Stack Algorithm **Usage of Perl Stack**

<pre> #!/usr/bin/perl -Wall local (\$line, @stack, \$stack, \$tmp); \$tos = 0; while (\$line = <STDIN>) { chop(\$line); if (\$line eq "tos") { print(\$stack{\$tos}); } elsif (\$line eq "pop") { print(\$stack{\$tos}); \$tos = \$tos - 1; } elsif (\$line eq "push") { \$tos = \$tos + 1; \$line = <STDIN>; chop(\$line); \$stack{\$tos} = \$line; } elsif (\$line eq "swap") { \$tmp = \$stack{\$tos}; \$stack{\$tos} = \$stack{\$tos-1}; \$stack{\$tos-1} = \$tmp; } } </pre>	<pre> push 3 tos 3 push 4 tos 4 pop 4 tos 3 </pre>
---	--

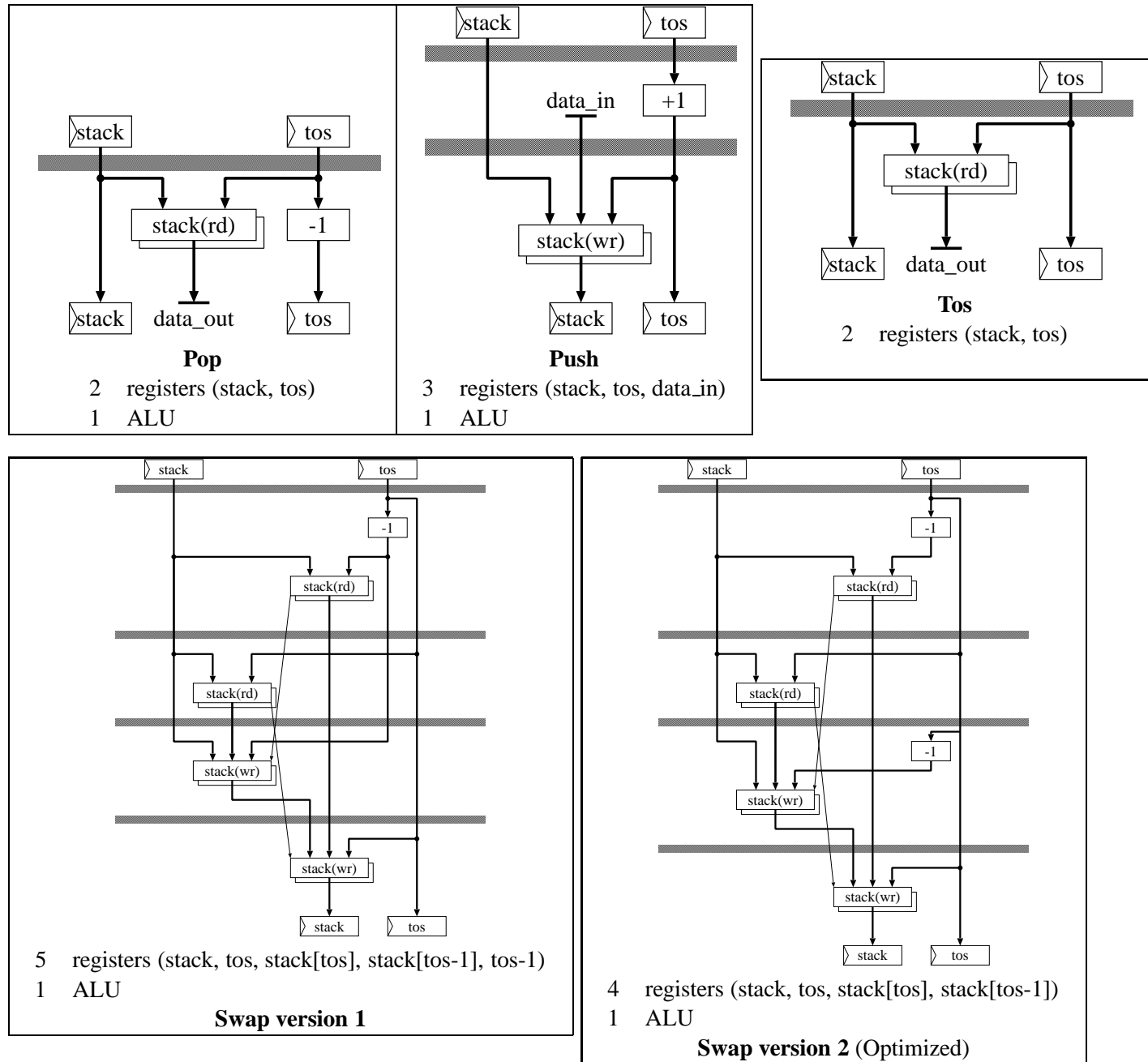
3.12.3 Stack: Dataflow Diagram

3.12.3.1 Initial Diagrams

Do one diagram for each operation. Do the initial dataflow diagrams without any clock cycle information.



3.12.3.2 Partition into Clock Cycles



3.12.4 Stack: High-Level Model

This high-level model is taken directly from the dataflow diagrams and block diagrams.

There is one process that combines control, datapath, and storage; except for the output (outp), which is done with a concurrent assignment statement.

Notice that there is a `next init` when `(reset = '1')`; after **every** wait statement. This is needed to get the circuit back to its initial state in the next clock cycle when reset is asserted.

First, we'll see the overall structure of the hlm architecture, and then the gory details.

architecture hlm of stack is

 ...declarations...

begin

 process

 begin

 init : loop

 ...reset assignments...

 loop

 wait until rising_edge(clk);

 next init when (reset = '1');

 case inp is

 when pop =>

 ...pop code...

 when push =>

 ...push code...

 when swap =>

 ...swap code...

 when tos =>

 ...tos code...

 when others =>

 next init;

 end case;

 end loop;

 end loop;

end process;

 outp <= stack(to_integer(tos));

end hlm;

Now for the actual code.

```
architecture hlm of stack is
    -----
    subtype data_ty is std_logic_vector(3 downto 0);
    type    stack_ty is array (15 downto 0) of data_ty;
    -----

    signal tos : unsigned(3 downto 0);
    signal tmp1, tmp2 : data_ty;
    signal stack : stack_ty;
    signal empty : std_logic;
    -----
begin
    -----
    process
    begin
        init : loop
            -----
            tos    <= to_unsigned(0,4);
            empty <= '1';
            -----

            loop
                -----
                wait until rising_edge(clk);
                next init when (reset = '1');
                -----

                case inp is
                    when pop =>
                        tos    <= tos - 1;
                    when push =>
                        if (empty = '0') then
                            tos <= tos + 1;
                        end if;
                        -----
                        wait until rising_edge(clk);
                        next init when (reset = '1');
                        -----

                        stack(to_integer(tos)) <= inp;
                        empty <= '0';
                    end case;
                end loop;
            end process;
        end architecture;
```

Continued...

...continued

```

    when swap =>
        tmp1 <= stack(to_integer(tos-1));
        -----
        wait until rising_edge(clk);
        next init when (reset = '1');
        -----
        tmp2 <= stack(to_integer(tos));
        -----
        wait until rising_edge(clk);
        next init when (reset = '1');
        -----
        stack(to_integer(tos-1)) <= tmp2;
        -----
        wait until rising_edge(clk);
        next init when (reset = '1');
        -----
        stack(to_integer(tos)) <= tmp1;
    when tos =>
        null;
    when others =>
        next init;
    end case;
end loop;
end loop;
end process;
-----
outp <= stack(to_integer(tos));
-----
end hlm;

```

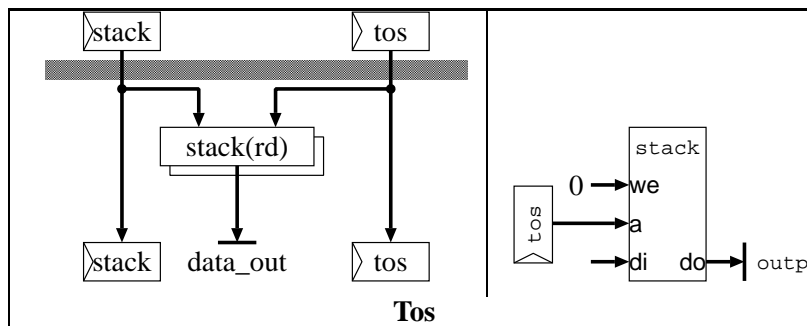
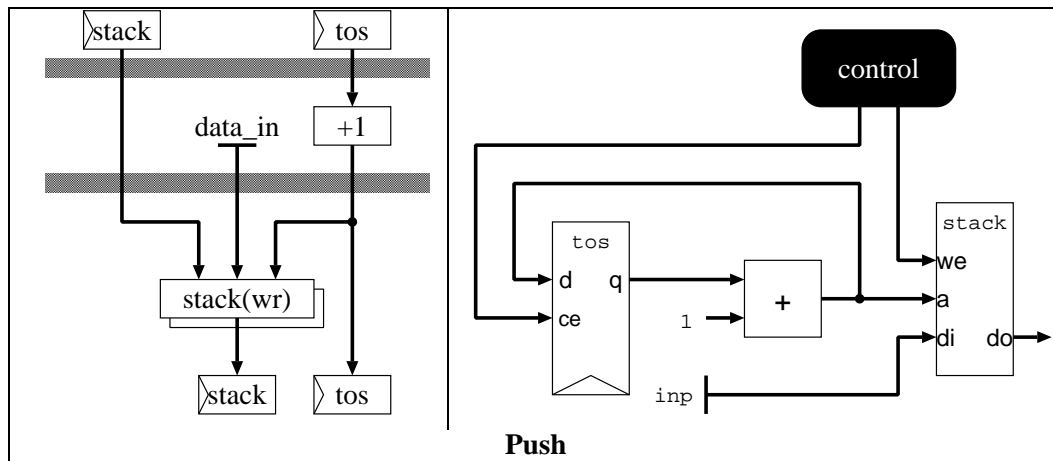
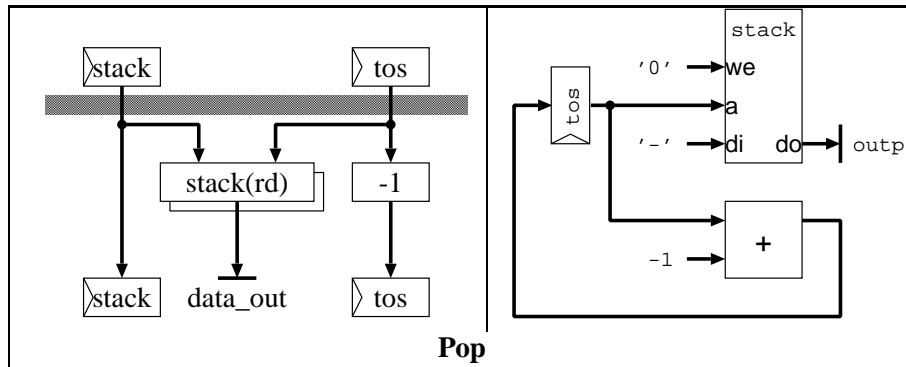
The high-level model is synthesizable, but might be large and slow.

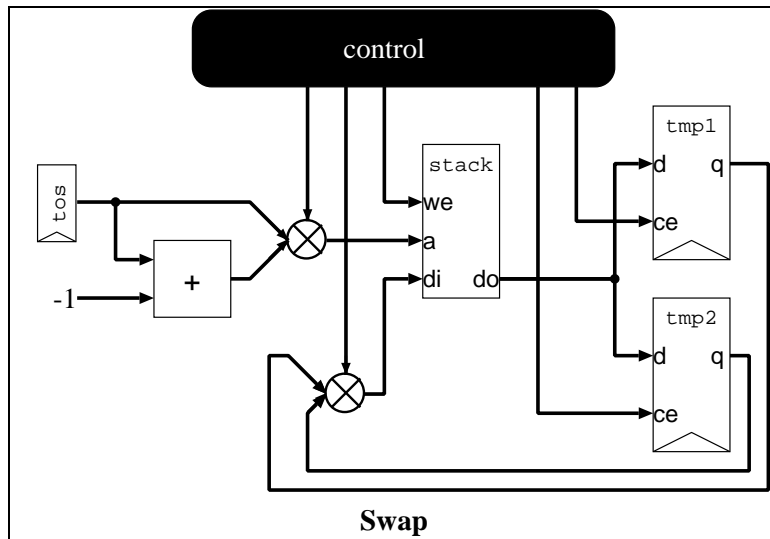
- It uses a 2-d array for the stack, rather than specialized memory components from the library.
- We are relying on the synthesis tool to build a state machine to drive the datapath. Sometimes, by writing code that is closer to gate-level hardware, we can improve performance and/or area.

3.12.5 Stack: Block Diagram

3.12.5.1 Individual Block Diagrams

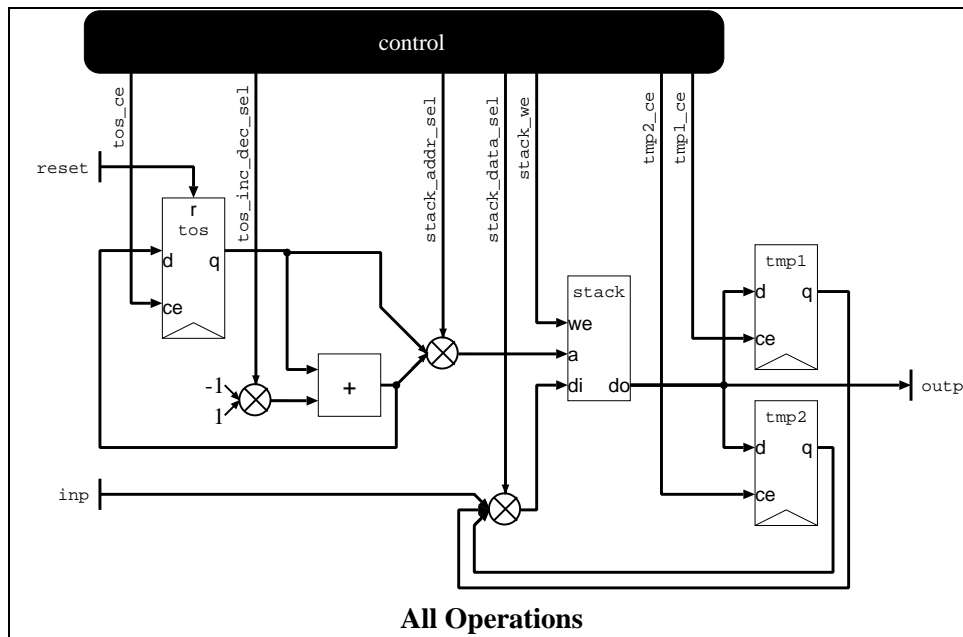
Build one block diagram for each operation.





3.12.5.2 Complete Block Diagram

Merge all of the block diagrams together, reusing components wherever possible.

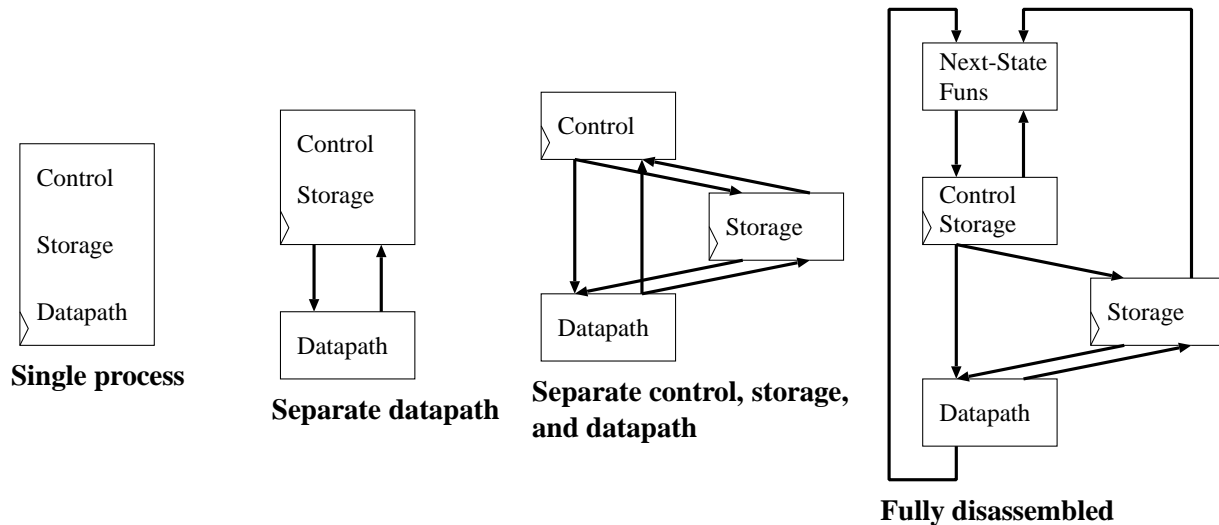


3.12.6 Stack: State Machine

3.12.7 Stack: Register Transfer Level

Structuring RTL Code

There are four different ways to structure your RTL code:



Section 1.8.4 described a variety of options for coding the individual modules in the above diagram. For example: whether to use both flopped and combinational signals, the number of target signals per process, and whether to `if` or `wait` statements for flip flops.

Stack RTL

To write the RTL code for the stack, consider the following options:

- Replacing the stack as an array with a component instantiation of a memory array from the FPGA libraries
- Defining a state machine and signals to control the datapath

(e.g. define a state type and a signal of type state and do assignments to current and next-state signals
Question to ponder: does an explicit state machine result in “better” hardware?)

3.12.7.1 Stack: Separate Control, Datapath and Storage

This design is derived directly from the hardware block diagram.

We separate the state machine and datapath using the control signals that drive the datapath (mux select lines, chip enables, etc).

The state machine drives signals that control the datapath.

The state machine is very similar to that in the high level model.

In every state we assign values to the signals that control the datapath.

The datapath is done with concurrent statements. By using concurrent statements, rather than processes, for the datapath, we eliminate the need for the datapath assignments to have sensitivity lists, which simplifies the code.

This style works best when there are a large number of states and a small number of datapath components.

Supplemental material available online (stack)

```
architecture sepfsm of stack is
    ...declarations...
begin
    ...component instantiation for memory...
    ...clocked process for state machine...
    ...clocked process for tmp1...
    ...clocked process for tmp2...
    ...clocked process for tos...
    ...concurrent assignment for tos_adj...
    ...concurrent assignment for stack_addr...
    ...concurrent assignment for stack_data_in...
end sepfsm;

architecture sepfsm of stack is
    signal tos,
           tos_adj,
           stack_addr      : unsigned(3 downto 0);
    signal inp_intern,
           stack_data_in,
           stack_data_out,
           tmp1,
           tmp2            : std_logic_vector(3 downto 0);
    signal synch_reset,
           empty,
           tos_inc_dec_sel,
           stack_addr_sel,
           tos_ce,
           stack_we,
           tmp1_ce,
           tmp2_ce         : std_logic;
    signal stack_data_sel  : std_logic_vector(1 downto 0);
    ...ram component instantiation...
```

Continued...

...continued

```

process begin
  init : loop
    -----
    empty          <= '1';
    tos_inc_dec_sel <= '-';
    stack_addr_sel  <= '-';
    tos_ce          <= '1';
    stack_we        <= '0';
    stack_data_sel  <= "--";
    tmp1_ce         <= '-';
    tmp2_ce         <= '-';
    -----

  loop
    -----
    wait until rising_edge(clk);
    next init when (reset = '1');
    -----

    case inp is
      when pop =>
        tos_inc_dec_sel <= '0';
        stack_addr_sel  <= '1';
        tos_ce          <= '1';
        stack_we        <= '0';
        stack_data_sel  <= "--";
        tmp1_ce         <= '-';
        tmp2_ce         <= '-';
      when push =>
        if (empty = '1') then
          tos_inc_dec_sel <= '-';
          stack_addr_sel  <= '0';
          tos_ce          <= '0';
        else
          tos_inc_dec_sel <= '1';
          stack_addr_sel  <= '1';
          tos_ce          <= '1';
        end if;
        stack_data_sel  <= "--";
        stack_we        <= '0';
        tmp1_ce         <= '-';
        tmp2_ce         <= '-';
        -----
        wait until rising_edge(clk);
        next init when (reset = '1');
        -----
        empty <= '0';  ...more assignments...
      when swap =>
        ...
    end case;
  end loop;
end loop;
end process;

```

Continued...

...continued

```

-----
process (clk)
begin
    if rising_edge(clk) then
        if (tmp1_ce = '1') then
            tmp1 <= stack_data_out;
        end if;
    end if;
end process;
... tmp2 assignment ...
-----

process (clk)
begin
    if rising_edge(clk) then
        if (reset = '1') then
            tos <= to_unsigned(0, 4);
        elsif (tos_ce = '1') then
            tos <= tos_adj;
        end if;
    end if;
end process;

-----
tos_adj <=  tos + 1 when (tos_inc_dec_sel = '1')
           else tos - 1 ;

...
...tos_adj, stack_addr, and stack_data_in...
end sepfsm;

```

3.12.7.2 **Stack: Datapath Operations**

The state machine in Section 3.12.7.1 controlled each datapath component individually.

An alternative style is for the state machine to tell the datapath what state it is in, or what global collection of operations to perform, then each part of the datapath decodes this and takes the appropriate action.

This style works best when there are a small number of states and a large number of datapath components.

architecture dp_op of stack is

```

-----
-- define the states
type dp_op_ty is
    (init_op,
     pop_op,
     push1_op,
     push2_op,
     swap_wr_tmp1_op,
     swap_wr_tmp2_op,
     swap_rd_tmp1_op,
     swap_rd_tmp2_op,
     nop_op
    );
signal dp_op : dp_op_ty;
signal tos,
       tos_adj,
       stack_addr      : unsigned(3 downto 0);
signal inp_intern,
       stack_data_in,
       stack_data_out,
       tmp1,
       tmp2            : std_logic_vector(3 downto 0);
signal empty,
       stack_we        : std_logic;
begin

```

Continued ...

...continued

```

-----
process
begin
  init : loop
    -----
    empty <= '1';
    dp_op <= init_op;
    loop
      -----
      wait until rising_edge(clk);
      next init when (reset = '1');
      -----
      case inp is
        when pop =>
          dp_op <= pop_op;
        when push =>
          dp_op <= push1_op;
          -----
          wait until rising_edge(clk);
          next init when (reset = '1');
          -----
          -- stack(to_integer(tos)) <= inp;
          dp_op <= push2_op;
          empty <= '0';
        when swap =>
          ...
          ...
      end case;
    end loop;
  end loop;
end process;

-----

process (clk)
begin
  if rising_edge(clk) then
    inp_intern <= inp;
  end if;
end process;

```

Continued...

...continued

```

-----
process (clk)
begin
  if rising_edge(clk) then
    if (dp_op = init_op) then
      tos <= to_unsigned(0,4);
    elsif ( (dp_op = pop_op)
            OR (dp_op = pushl_op and (empty = '0'))
          )
      then
        tos <= tos_adj;
      end if;
    end if;
  end process;
-----
tos_adj
  <=  tos + to_unsigned(1,3)  when (dp_op = pushl_op)
    else tos - to_unsigned(1,3)
;
-----

stack_addr
  <= tos_adj
    when
      ( (dp_op = pop_op)
        OR ((dp_op = pushl_op) AND (empty = '0'))
        OR (dp_op = swap_wr_tmp1_op)
        OR (dp_op = swap_rd_tmp2_op)
      )
    else tos
;

...stack_data_in, stack_we, out, ram ...

end dp_op;

```

3.12.7.3 Stack: Explicit State Machine

Here we drop the `loop ... wait ...` style of implicit state machines and build an explicit state machine with current and next state signals.

Notice that the stack is such a simple design that each datapath operation in the Dp-Op architecture is used in only one state. This is a sign that the Dp-Op style is not well-suited to the stack.

This example also illustrates the use of a function to capture common code. The function is used here to determine which state to go to next when a new input instruction arrives.

architecture state of stack is

```

type state_ty is
    (init_st,
     pop_st,
     push1_st, push2_st,
     swap_wr_tmp1_st, swap_wr_tmp2_st,
     swap_rd_tmp1_st, swap_rd_tmp2_st,
     nop_st
    );
signal state, state_n : state_ty;
...
...
-----
function restart
    (inp : std_logic_vector(3 downto 0))
    return state_ty
is
begin
    case inp is
        when pop =>
            return(pop_st);
        when push =>
            return(push1_st);
        when swap =>
            return(swap_wr_tmp1_st);
        when others =>
            return(nop_st);
    end case;
end restart;
begin
    -----
    process (clk) begin
        if rising_edge(clk) then
            if (reset = '1') then
                state    <= init_st;
                empty_n <= '1';
            else
                state    <= state_n;
                empty_n <= empty;
            end if;
        end if;
    end process;

```

Continued...

...continued

```

-----
process (state, inp) begin
  case state is
    when init_st | pop_st | push2_st
      | swap_wr_tmp2_st | nop_st
      =>
        state_n <= restart(inp);
    when push1_st =>
        state_n <= push2_st;
    when swap_rd_tmp1_st =>
        state_n <= swap_rd_tmp2_st;
    when swap_rd_tmp2_st =>
        state_n <= swap_wr_tmp1_st;
    when swap_wr_tmp1_st =>
        state_n <= swap_wr_tmp2_st;
  end case;
end process;
...

process (clk)
begin
  if rising_edge(clk) then
    if (state = init_st) then
      tos <= to_unsigned(0,4);
    elsif ( (state = pop_st)
            OR (state = push1_st and (empty = '0'))
          )
    then
      tos <= tos_adj;
    end if;
  end if;
end process;

-----
tos_adj
  <=  tos + to_unsigned(1,3)  when (state = push1_st)
    else tos - to_unsigned(1,3)
;

-----
stack_addr <=  tos_adj when
  ( (state = pop_st)
    OR ((state = push1_st) AND (empty = '0'))
    OR (state = swap_wr_tmp1_st)
    OR (state = swap_rd_tmp2_st)
  )
  else tos ;
...
end state;

```

3.13 Design Problems

P3.1 Synthesis

This question is about using VHDL to implement memory structures on FPGAs.

P3.1.1 Data Structures

If you have to write your own code (i.e. you do not have a library of memory components or a special component generation tool such as LogiBlox or CoreGen). What datastructures in VHDL would you use when creating a register file?

P3.1.2 Own Code vs Libraries

When using VHDL for an FPGA, under what circumstances is it better to write your own VHDL code for memory, rather than instantiate memory components from a library?

P3.2 Design Guidelines

While you are grocery shopping you encounter your co-op supervisor from last year. She's now forming a startup company in Waterloo that will build digital circuits. She's writing up the design guidelines that all of their projects will follow. She asks for your advice on some potential guidelines.

What is your response to each question?

What is your justification for your answer?

What are the tradeoffs between the two options?

0. **Sample** Should all projects use **silicon** chips, or should all use **biological** chips, or should each project choose its own technique?

Answer: *All projects should use silicon based chips, because biological chips don't exist yet. The tradeoff is that if biological chips existed, they would probably consume less power than silicon chips.*

1. Should all projects use an **asynchronous reset** signal, or should all use a **synchronous reset** signal, or should each project choose its own technique?
2. Should all projects use **latches**, or should all projects use **flip-flops**, or should each project choose its own technique?
3. Should all *chips* have **registers on the inputs and outputs** or should chips have the **inputs and outputs directly connected to combinational circuitry**, or should each project choose its own technique? By "register" we mean either flip-flops or latches, based upon your answer to the previous question. If your answer is different for inputs and outputs, explain why.

- you may change the times when signals are read from the environment
- you may **not** increase the resource usage (input ports, registers, output ports, f components, g components)
- you may **not** increase the clock period

P3.4 Dataflow Diagram Design

Your manager has given you the task of implementing the following pseudocode in an FPGA:

```
if is_odd(a + d)
    p = (a + d)*2 + ((b + c) - 1)/4;
else
    p = (b + c)*2 + d;
```

- NOTES:
- 1) **You must use registers on all input and output ports.**
 - 2) p , a , b , c , and d are to be implemented as 8-bit signed signals.
 - 3) A 2-input 8-bit ALU that supports both addition and subtraction takes 1 clock cycle.
 - 4) A 2-input 8-bit multiplier or divider takes 4 clock cycles.
 - 5) A small amount of additional circuitry (e.g. a NOT gate, an AND gate, or a MUX) can be squeezed into the same clock cycle(s) as an ALU operation, multiply, or divide.
 - 6) You can require that the environment provides the inputs in any order and that it holds the input signals at the same value for multiple clock cycles.

P3.4.1 Maximum performance

What is the minimum number of clock cycles needed to implement the pseudocode with a circuit that has two input ports?

What is the minimum number of ALUs, multipliers, and dividers needed to achieve the minimum number of clock cycles that you just calculated?

P3.4.2 Minimum area

What is the minimum number of datapath storage registers (8, 6, 4, and 1 bit) and clock cycles needed to implement the pseudocode if the circuit can have at most one ALU, one multiplier, and one divider?

P3.5 Design and Optimization

Design a circuit that performs the following operation: $P = (a+d) + ((b - c) - 1)$

Optimize your design for area.

P3.6 Dataflow Diagrams with Memory Arrays

Component	Delay
Register	5 ns
Adder	25 ns
Subtractor	30 ns
ALU with +, −, >, =, −, AND, XOR	40 ns
Memory read	60 ns
Memory write	60 ns
Multiplication	65 ns
2:1 Multiplexor	5 ns

NOTES:

1. The inputs of the algorithms are a and b .
2. The outputs of the algorithms are p and q .
3. You must register both your inputs and outputs.
4. You may choose to read your input data values at any time and produce your outputs at any time. For your inputs, you may read each value only once (i.e. the environment will not send multiple copies of the same value).
5. Execution time is measured from when you read your first input until the latter of producing your last output or the completion of writing a result to memory
6. M is an internal memory array, which must be implemented as dual-ported memory with one read/write port and one write port.
7. M supports synchronous write and asynchronous read.
8. Assume all memory address and other arithmetic calculations are within the range of representable numbers (i.e. no overflows occur).
9. If you need a circuit not on the list above, assume that its delay is 30 ns.
10. You may sacrifice area efficiency to achieve high performance, but marks will be deducted for extra hardware that does not contribute to performance.

P3.6.1 Algorithm 1**Algorithm**

```

q = M[b];
M[a] = b;
p = (M[b-1]) * b) + M[b];

```

Assuming $a > b$, draw a dataflow diagram that is optimized for the fastest overall execution time.

P3.6.2 Algorithm 2

```

q = M[b];
M[a] = b;
p = (M[b-1]) * b) + M[b];

```

Assuming $a \leq b$, draw a dataflow diagram that is optimized for the fastest overall execution time.

P3.7 2-bit adder

This question compares an FPGA and generic-gates implementation of 2-bit full adder.

P3.7.1 Generic Gates

Show the implementation of a 2 bit adder using NAND, NOR, and NOT gates.

P3.7.2 Xilinx FPGA

Show the CLB implementation of a 2 bit adder in a Xilinx Spartan XCS10 FPGA by drawing the schematic of a CLB and showing the equations for the lookup tables.

P3.8 Sketches of Problems

1. calculate resource usage for a dataflow diagram (input ports, output ports, registers, datapath components)
2. calculate performance data for a dataflow diagram (clock period and number of cycles to execute (CPI))
3. given a dataflow diagram, calculate the clock period that will result in the optimum performance
4. given an algorithm, design a dataflow diagram
5. given a dataflow diagram, design the datapath and finite state machine
6. optimize a dataflow diagram to improve performance or reduce resource usage
7. given fsm diagram, pick VHDL code that “best” implements diagram — correct behaviour, simple, fast hardware — or critique hardware