5.9 Using a finite-state machine to control a simple cache

This section has been set as optional by your instructor.

We can now build control for a cache, just as we implemented control for the single-cycle and pipelined datapaths in COD Chapter 4 (The Processor). This section starts with a definition of a simple cache and then a description of *finite-state machines* (FSMs). It finishes with the FSM of a controller for this simple cache. COD Section 5.12 (Advanced material: Implementing cache controllers) goes into more depth, showing the cache and controller in a new hardware description language.

A simple cache

We're going to design a controller for a straightforward cache. Here are the key characteristics of the cache:

- Direct-mapped cache
- Write-back using write allocate
- Block size is four words (16 bytes or 128 bits)
- Cache size is 16 KiB, so it holds 1024 blocks
- 32-bit addresses
- The cache includes a valid bit and dirty bit per block

From COD Section 5.3 (The basics of caches), we can now calculate the fields of an address for the cache:

- Cache index is 10 bits
- Block offset is 4 bits
- Tag size is 32 - (10 + 4) or 18 bits

The signals between the processor to the cache are

- 1-bit Read or Write signal
- 1-bit Valid signal, saying whether there is a cache operation or not
- 32-bit address
- 32-bit data from processor to cache
- 32-bit data from cache to processor
- 1-bit Ready signal, saying the cache operation is complete

The interface between the memory and the cache has the same fields as between the processor and the cache, except that the data fields are now 128 bits wide. The extra memory width is generally found in microprocessors today, which deal with either 32-bit or 64-bit words in the processor while the DRAM controller is often 128 bits. Making the cache block match the width of the DRAM simplified the design. Here are the signals:

- 1-bit Read or Write signal
- 1-bit Valid signal, saying whether there is a memory operation or not
- 32-bit address
- 128-bit data from cache to memory
- 128-bit data from memory to cache
- 1-bit Ready signal, saying the memory operation is complete

Note that the interface to memory is not a fixed number of cycles. We assume a memory controller that will notify the cache via the Ready signal when the memory read or write is finished.

Before describing the cache controller, we need to review finite-state machines, which allow us to control an operation that can take multiple clock cycles.

PARTICIPATION ACTIVITY 5.9.1: A simple cache.

1) The number of cycles required to read from or write to the memory _____ .
   - ○ is fixed
   - ○ varies

2) The processor-cache interface and the memory-cache interface is _____ .
   - ○ the same
   - ○ different

Finite-state machines

To design the control unit for the single-cycle datapath, we used truth tables that specified the setting of the control signals based on the instruction class. For a cache, the control is more complex because the operation can be a series of steps. The control for a cache must specify both the signals to be set in any step and the next step in the sequence.

The most common multistep control method is based on *finite-state machines*, which are usually represented graphically. A finite-state machine consists of a set of states and directions on how to change states. The directions are defined by a *next-state function*, which maps the current state and the inputs to a new state. When we use a finite-state machine for control, each state also specifies a set of outputs that are asserted when the machine is in that state. The implementation of a finite-state machine usually assumes that all outputs that are not explicitly asserted are deasserted. Similarly, the correct operation of the datapath depends on the fact that a signal that is not explicitly asserted is deasserted, rather than acting as a don't care.

Finite-state machine: A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

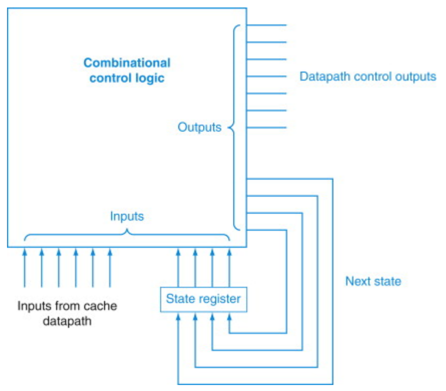
Next-state machine: A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

Multiplexor controls are slightly different, since they select one of the inputs, whether they are 0 or 1. Thus, in the finite-state machine, we always specify the setting of all the multiplexor controls that we care about. When we implement the finite-state machine with logic, setting a control to 0 may be the default and therefore may not require any gates. A simple example of a finite-state machine appears in COD Appendix A (The Basics of Logic Design), and if you are unfamiliar with the concept of a finite-state machine, you may want to examine COD Appendix A (The Basics of Logic Design) before proceeding.

A finite-state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the data-path signals to be asserted and the next state. The figure below shows how such an implementation might look. COD Appendix C (Mapping Control to Hardware) describes in detail how the finite-state machine is implemented using this structure. In COD Section A.3 (Combinational logic), the combinational control logic for a finite-state machine is implemented both with either a ROM (*read-only memory*) or a PLA (*programmable logic array*). (Also see COD Appendix A (The Basics of Logic Design) for a description of these logic elements.)

Figure 5.9.1: Finite-state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state (COD Figure 5.38).

The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. Notice that in the finite-state machine used in this chapter, the outputs depend only on the current state, not on the inputs. We use color to indicate that these are control lines and logic versus data lines and logic. The *Elaboration* below explains this in more detail.



PARTICIPATION ACTIVITY 5.9.2: Implementing multistep control.

1) The cache controller is designed using a _____ .

- truth table
- finite-state machine

2) A finite-state machine consists of a combinational control logic block and a temporary register used to hold the _____ state.

- current
- next
- previous

Elaboration

Note that this simple design is called a blocking cache, in that the processor must wait until the cache has finished the request. COD Section 5.12 (Advanced material: Implementing cache controllers) describes the alternative, which is called a nonblocking cache.

Elaboration

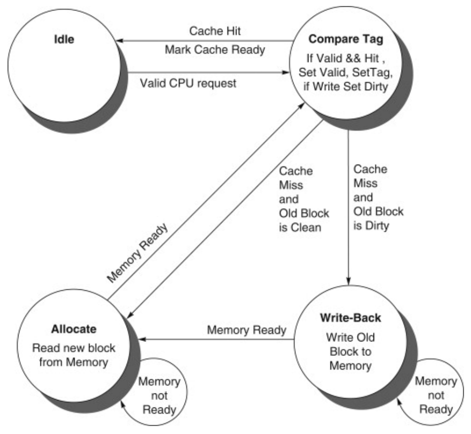
The style of finite-state machine in this book is called a Moore machine, after Edward Moore. Its identifying characteristic is that the output depends only on the current state. For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has just the next-state output.

An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are needed early in the clock cycle, do not depend on the inputs, but only on the current state. In COD Appendix A (The Basics of Logic Design), when the implementation of this finite-state machine is taken down to logic gates, the size advantage can be clearly seen. The potential disadvantage of a Moore machine is that it may require additional states. For example, in situations where there is a one-state *difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.*

FSM for a simple cache controller

The figure below shows the four states of our simple cache controller:

Figure 5.9.2: Four states of the simple controller (COD Figure 5.39).


Idle
Cache Hit
Mark Cache Ready
Valid CPU request
Compare Tag
If Valid && Hit , Set Valid, SetTag, if Write Set Dirty
Cache Miss and Old Block is Clean
Cache Miss and Old Block is Dirty
Memory Ready
Allocate
Read new block from Memory
Memory not Ready
Memory Ready
Write-Back
Write Old Block to Memory
Memory not Ready

- *Idle*: This state waits for a valid read or write request from the processor, which moves the FSM to the Compare Tag state.
- *Compare Tag*: As the name suggests, this state tests to see if the requested read or write is a hit or a miss. The index portion of the address selects the tag to be compared. If the data in the cache block referred to by the index portion of the address are valid, and the tag portion of the address matches the tag, then it is a hit. Either the data are read from the selected word if it is a load or written to the selected word if it is a store. The Cache Ready signal is then set. If it is a write, the dirty bit is set to 1. Note that a write hit also sets the valid bit and the tag field; while it seems unnecessary, it is included because the tag is a single memory, so to change the dirty bit we likewise need to change the valid and tag fields. If it is a hit and the block is valid, the FSM returns to the idle state. A miss first updates the cache tag and then goes either to the Write-Back state, if the block at this location has dirty bit value of 1, or to the Allocate state if it is 0.
- *Write-Back*: This state writes the 128-bit block to memory using the address composed from the tag and cache index. We remain in this state waiting for the Ready signal from memory. When the memory write is complete, the FSM goes to the Allocate state.
- *Allocate*: The new block is fetched from memory. We remain in this state waiting for the Ready signal from memory. When the memory read is complete, the FSM goes to the Compare Tag state. Although we could have gone to a new state to complete the operation instead of reusing the Compare Tag state, there is a good deal of overlap, including the update of the appropriate word in the block if the access was a write.

This simple model could easily be extended with more states to try to improve performance. For example, the Compare Tag state does both the compare and the read or write of the cache data in a single clock cycle. Often the compare and cache access are done in separate states to try to improve the clock cycle time. Another optimization would be to add a write buffer so that we could save the dirty block and then read the new block first so that the processor doesn't have to wait for two memory accesses on a dirty miss. The cache would then write the dirty block from the write buffer while the processor is operating on the requested data.

COD Section 5.12 (Advancing material: Implementing cache controllers) goes into more detail about the FSM, showing the full controller in a hardware description language and a block diagram of this simple cache.

PARTICIPATION ACTIVITY 5.9.3: A simple cache controller.

Refer to the figure above (COD Figure 5.39 (Four states of the simple controller)).

1) The first state fetches a new cache block from memory.
   - ○ True
   - ○ False

2) The Memory Ready signal is set if the requested read or write is a hit.
   - ○ True
   - ○ False

3) The performance of the simple controller can be improved by incorporating additional states.
   - ○ True
   - ○ False

Provide feedback on this section