Disclaimer: The contents of this document are scribe notes for The University of Texas at Austin EE360N Fall 2008, Computer Architecture. The notes capture the class discussion and may contain erroneous and unverified information and comments.

Implementing Another ISA, Basic Pipelining

Lecture #4: September 10, 2008
Lecturer: Derek Chiou
Scribe: Sandeep Gupta and Terrence Lynch

1 Recap and Outline

In lecture three, we talked about microcode. The first part of this lecture concludes the microcode discussion, emphasizing the relation between the LC-3b state machine and the memory-based lookup table that implements the steps in the instruction cycle.

Next we looked at a stack machine instruction set architecture (ISA) implementation.

In the remainder of the class, we took our first look at pipelining.

2 Microcode Engine Control

The LC-3b ISA is implemented in hardware using a complex state machine. Each state has a unique set of inputs and outputs that determine the next state and the control signals on the datapath.

Operating in concert with this state machine is a memory-based lookup table, alternately called the microcode control store or nanocode. It may be useful to consider each use of the control store as a sub-routine call by the LC-3b state machine. The microcode controls the dataflow on the bus.

For example, assume the LC-3 is in the process of fetching an instruction. The instruction just became available in the memory data register (MDR), and the state transitions to loading the MDR into the instruction register (IR). The microcode control store contains bits that drive the tri-state buffers on the bus, placing the new instruction on the bus and into the IR.

The microcode control store has a total of several inputs and many outputs that determine, with the help of some combinational logic and a microcode program counter,

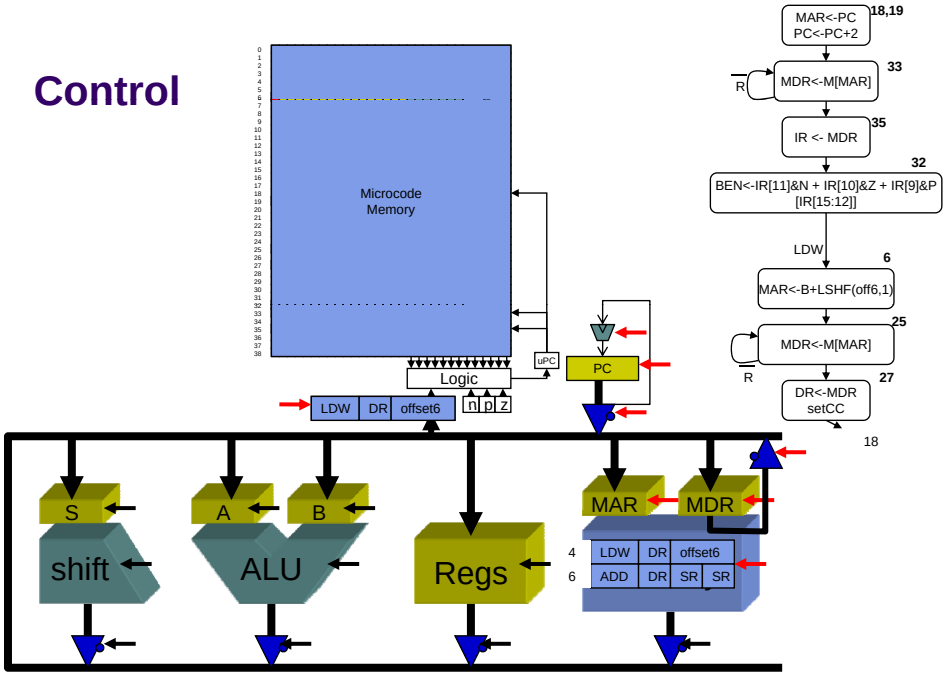

Figure 1: Architecture of the Microcode Control Store

the next state of the microcode. Once the microcode has cycled through the steps to complete a specific instruction, the microcode state machine fetches the next instruction and processes it.

Why spend the time developing microcode when it is easier to implement an ISA in software? The answer lies in the amount of resources required to implement an ISA under the two scenarios.

Given $NumInstInProgram$ is the number of instructions in a program, $\frac{Num\mu Ops}{inst}$ is the number of microcode operations per instruction, $\frac{bits}{\mu Op}$ is the number of bits per microcode, and $(\frac{bits}{inst})$ is the number of bits per instruction, then the amount of memory required for a program without microcode is

$$NumInstrInPrgm \times (\frac{num\mu Ops}{inst}) \times (\frac{bits}{\mu Op})$$

Adding microcode reduces the amount of memory required:

$$NumInstrInPrgm \times (\frac{bits}{inst}) + NumInstInISA \times (\frac{num\mu Ops}{inst}) \times (\frac{bits}{\mu Op})$$

An alternate view of microcode is to think of the function it performs as a subroutine call. That is, only one copy of a microcode instruction is needed. Without microcode, a

new copy of the microcoded instruction would need to placed in a program every time that the function is needed. So, the number of instructions in a program is significantly reduced when a microcoded engine is used.

3 Implementing an Alternate ISA

An ISA and processor work closely together and have interdependencies, but that does not mean every microarchitecture-data path combination is unique. The control store will certainly be different, but the actual devices and connections in the datapath can be fairly similar. A stack machine ISA illustrates this point well.

3.1 Stack Machine Review

A stack machine is called a zero address machine because operations occur without addresses or arguments. For example the arguments for an ADD instruction are implicit: they occupy the first two locations in a stack of consecutive memory locations.

Let's look closer at how two numbers are added together in a stack machine. The instructions PUSH M and POP M allow data in general memory to be moved to and from the stack for manipulation. PUSH places the contents of memory location M on the top of the stack, figuratively pushing the other values on the stack one location lower. This is actually accomplished using a stack pointer. Data from memory is moved to the next spot of memory on top of the stack, and the stack pointer is incremented. So, in reality, the other items on the stack do not move, but their position changes relative to the stack pointer.

The POP instruction is a PUSH in reverse. The stack pointer is pointing to the data at the top of the stack. So the POP instruction takes that data, stores it in memory location M, and then adjusts the stack pointer, to the next lowest data on the stack.

The ADD instruction takes the top two pieces of data on the stack, adds them together, removes the two arguments from the stack and places the result on the top of the stack.

So, to add two numbers located in M1 and M2 and store the result in M3, the instructions are:

- PUSH M1
- PUSH M2
- ADD
- POP M3

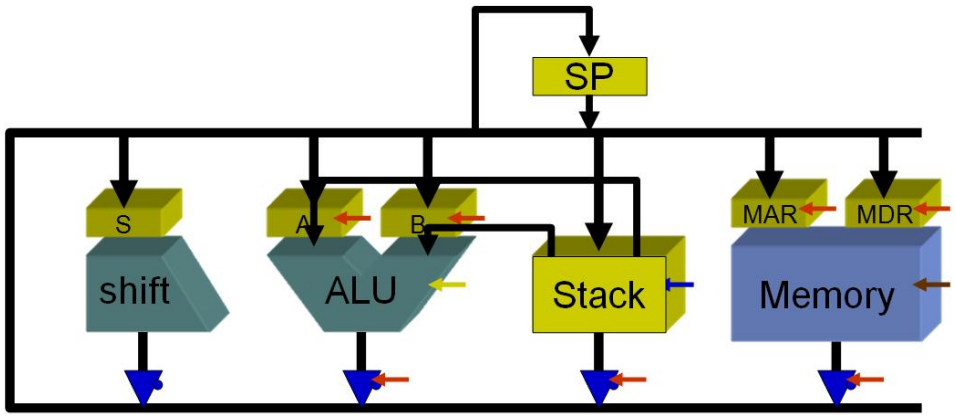

Figure 2: The Datapath for a Stack Machine

3.2 The Stack Machine on a Microcoded Engine

What are the differences, then, between the microcoded engine and datapath of a stack machine and that of a three address machine like the LC-3b (see Figure 2 for the datapath of a stack machine)?

1. The stack pointer is a discreet element of the datapath in a stack machine. There is no corresponding element in the three address machine
2. The register file on the three address machine is replaced by the stack, but they do the same thing: serve as temporary storage during operations.
3. There is no need for the inputs to the arithmetic and logic unit (ALU) to be on the bus in the stack machine, because its operations require no addresses. There is a direct path from the stack to the ALU.

The stack machine is simple and elegant - it uses fewer bits, so code density increases. If memory is expensive, then a stack machine is a nice alternative to a three address machine. But the stack machine limits addressable memory space because the top four bits of each instruction are used for the opcode. On a multi-address machine, the entire 16 bits can be used for addressing. A stack machine's addressable memory can be expanded using either variable instruction sizing or pitfall-laden self-modifying code.

4 Introduction to Pipelining

4.1 Basic Pipelining

Consider an individual ADD instruction for the LC-3b. To execute an ADD, the LC-3b state machine goes through the six phases in the instruction cycle: fetch, decode,

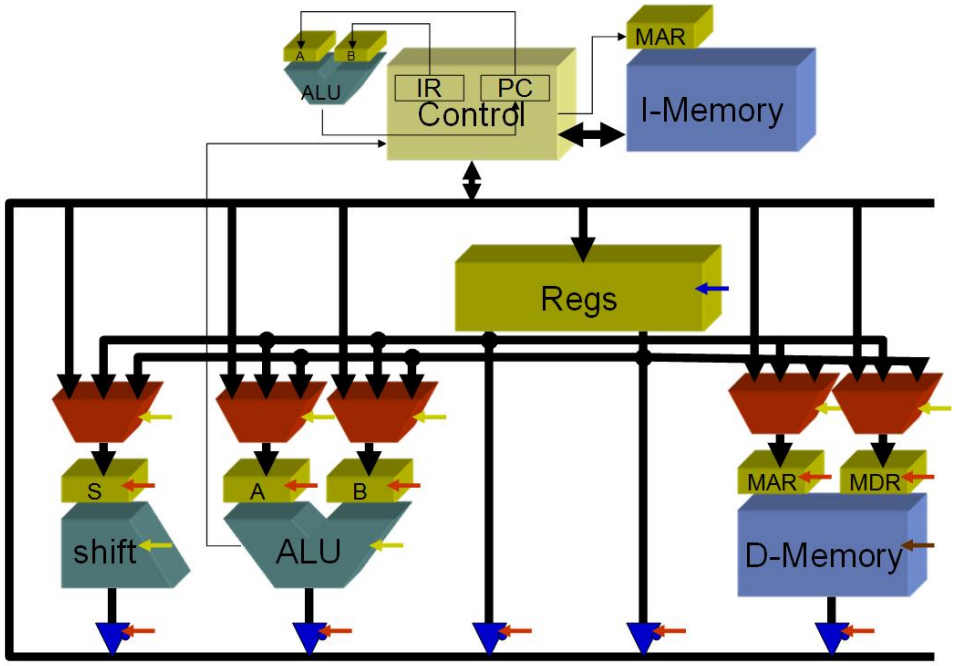

Figure 3: Basic Two-Stage Pipelining without Dependencies

evaluate, fetch operands, execute, and store result. Up to now, we have only considered a single ported register file, which requires each instruction to complete before another can be initiated.

In the case of the ADD instruction, the bus connection between the register file and the ALU sits idle while the result is returned to the register file after the ALU completes its operation. See Figure 3.

If, however, a dual-ported memory file is used, then successive operations that have no dependencies can be pipelined to the ALU. In this case the first port of the register file sends the operands of the first instruction to the ALU and on the next cycle, the second port sends the next instruction. This happens at the same time the result of the first operation is returned to the register file. This is called a two stage pipeline. Pipelining is a mechanism used to optimize use of resources and increase the throughput and, ultimately, the performance of the system.

This basic model only works when no dependencies exist between the two instructions in the pipeline, as in the following case:

1. $C = A + B$
2. $F = D + E$.

Problems arise in the following situation

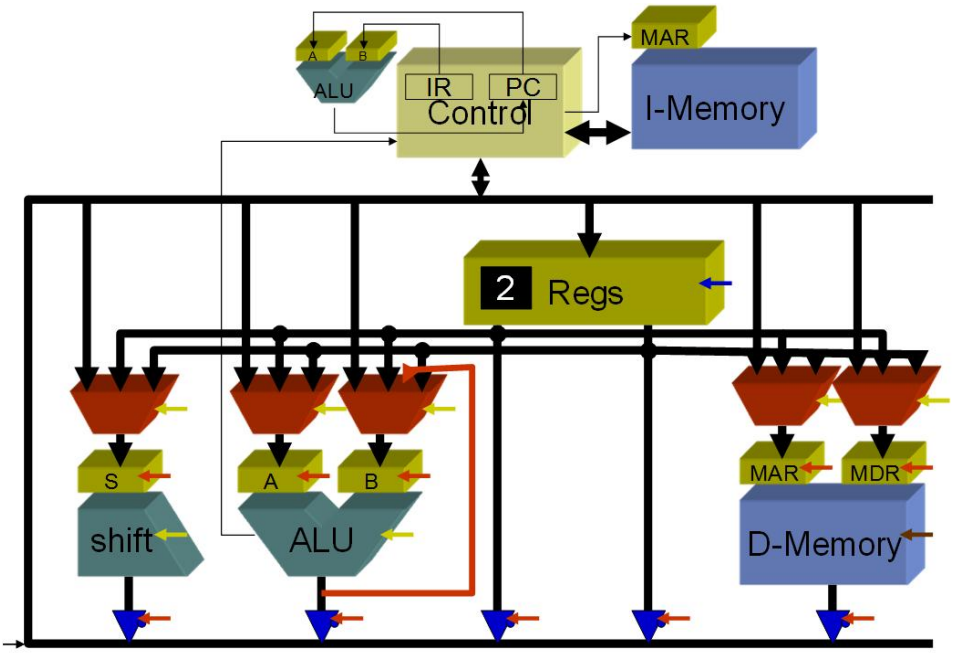

Figure 4: Basic Two-Stage Pipelining with Dependencies

1. $C = A + B$
2. $F = D + C$

because there is a dependency between the two ADD instructions. The value in the result of A+B = C has not yet been returned to the register file and so the next operation will not correctly execute if it is pipelined.

To solve this problem, the computer architect must add dependency checking to the microcode. Some possible solutions for dependencies are:

- Bypass: As in Figure 4, the result of the current ALU operation may be returned one of the ALU inputs. This ensures the updated value of is used in the next operation, but requires the addition of a mux to the datapath.
- Interlocks: if the checker finds a dependency between consecutive instructions, it prevents pipelining from occurring until the dependency is no longer and issue.
- Reorder instructions: The microcode inserts an independent instruction between two dependent instructions.
- NOP: The no operation instruction inserts a dead cycle into the instruction flow, allowing the dependency to pass. This instruction is generated either by the programmer or the compiler.

Now that we know how to pipeline, why is it necessary? If there were no wasted resources in a computer, then pipelining would be unnecessary. Pipelining is only useful if resources are being underutilized. For example, in the case of the two ADD instructions, the register file was sitting idle while the ALU was computing the first ADD. In that case, pipelining enables both the register file and the ALU to both be working at the same time by processing the register read part of the second ADD at the same time as the ALU operation of the first ADD.

Pipelining increases the issue rate between instructions. In this example, without pipelining, an instruction is issued every two clock cycles, but with pipelining an instruction can be issued almost every clock cycle.

Lastly, pipelining potentially enables a faster clock rate.

4.2 Processor Metrics

There are two metrics often used by the computer architect to benchmark improvements in processor performance: instructions per clock cycle (IPC) and clock cycles per instruction (CPI). They are dependent on the application, the ISA, the micro-architecture, and the system.

CPI contributes one variable in the equation for the overall speed of a computer, which is determined as follows for a specific program:

$$TotalTime = TotalNumInst \times (\frac{clockCycles}{Inst}) \times (\frac{Time}{ClockCycle})$$

or

$$TotalTime = TotalNumInst \times CPI \times CycleTime.$$

So, the goal of the computer architect is to minimize the number of clock cycles per instruction and minimize the time between rising edges of the clock.

4.3 Trade-offs of Pipelining

Pipelining will certainly increase the performance of a computer, but taken to an extreme, the benefits diminish. For example, the register file still operates at a slower speed. Also, if a dependency occurs and the pipeline needs to stall to deal with it, the processing rate is temporarily cut in half. Sometimes an instruction is not easy to split up, so the pipeline is not completely efficient. Lastly, pipeline registers have delay.