ECE120: Introduction to Computer Engineering

Notes Set 4.1 Control Unit Design

Appendix C of the Patt and Patel textbook describes a microarchitecture for the LC-3 ISA, including a control unit implementation. In this set of notes, we introduce a few concepts and strategies for control unit design, using the textbook's LC-3 microarchitecture to help illustrate them. Several figures from the textbook are reproduced with permission in these notes as an aid to understanding.

The control unit of a computer based on the von Neumann model can be viewed as an FSM that fetches instructions from memory and executes them. Many possible implementations exist both for the control unit itself and for the resources that it controls, the other components in the von Neumann model, which we collectively call the **datapath**. In this set of notes, we discuss two strategies for structured control unit design and introduce the idea of using memories to encode logic functions.

Let's begin by recalling that the control unit consists of three parts: a high-level FSM that controls instruction processing, a program counter (PC) register that holds the address of the next instruction to be executed, and an instruction register (IR) that holds the current instruction as it executes.

Other von Neumann components provide inputs to the control unit. The memory unit, for example, contains the instructions and data on which the program executes. The processing unit contains a register file and condition codes (N, Z, and P for the LC-3 ISA). The outputs of the control unit are signals that control operation of the datapath: the processing unit, the memory, and the I/O interfaces. The basic problem that we must solve, then, for control unit design, is to map instruction processing and the state of the FSM (including the PC and the IR) into appropriate sequences of **control signals** for the datapath.

4.1.1 LC-3 Datapath Control Signals

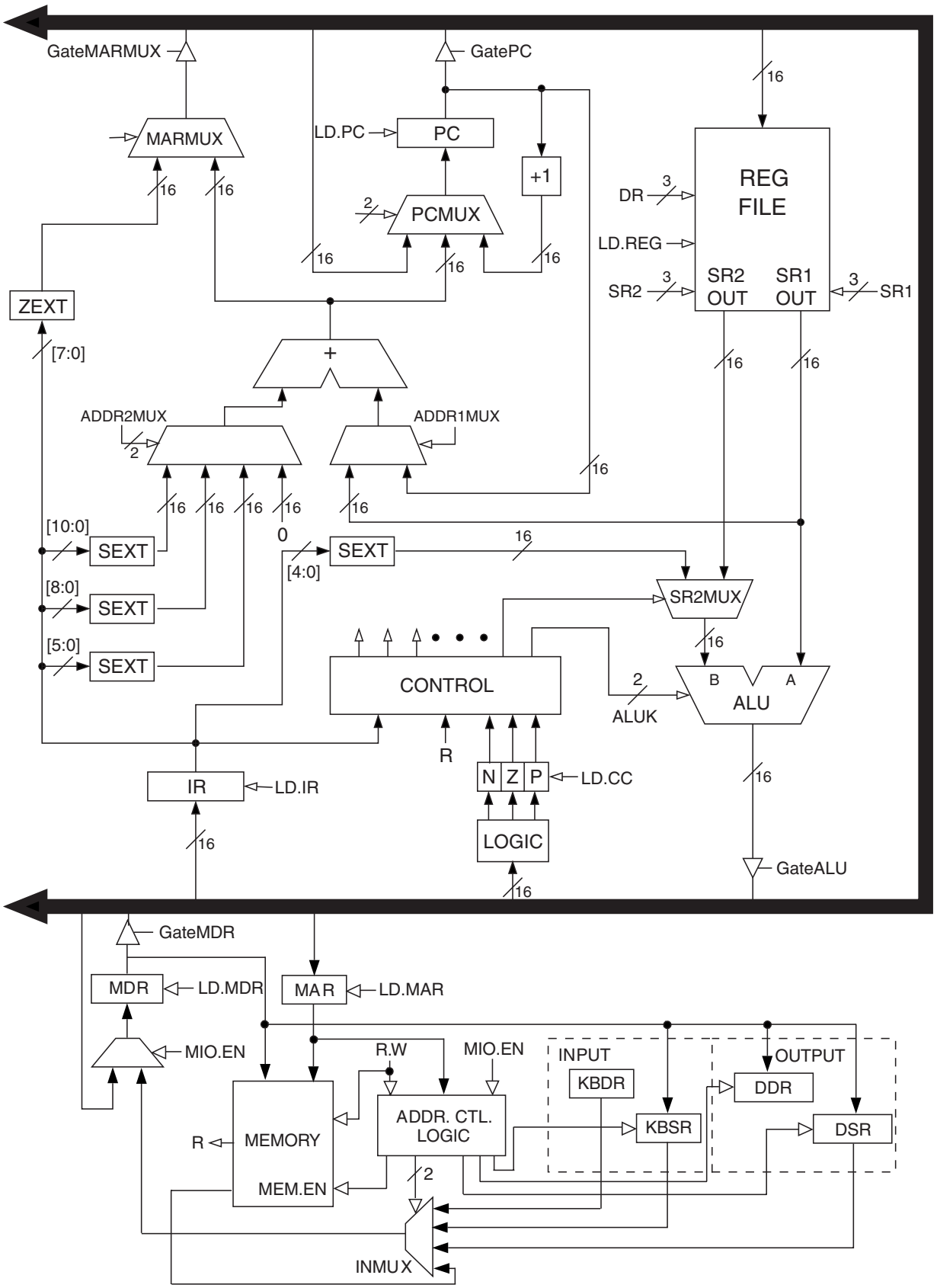
As we have skipped over the implementation details of interrupts and privilege of the LC-3 in our class, let's consider a microarchitecture and datapath without those capabilities. The figure on the next page (Patt and Patel Figure C.3) shows an LC-3 datapath and control signals without support for interrupts and privilege.

Some of the datapath control signals mentioned in the textbook are no longer necessary in the simplified design. Let's discuss the signals that remain and give some examples of how they are used. A list appears to the right.

signal	meaning
LD.MAR	load new value into memory address register
LD.MDR	load new value into memory data register
LD.IR	load new value into instruction register
LD.BEN	load new value into branch enable register
LD.REG	load new value into register file
LD.CC	load new values into condition code registers (N,Z,P)
LD.PC	load new value into program counter
GatePC	write program counter value onto bus
GateMDR	write memory data register onto bus
GateALU	write arithmetic logic unit result onto bus
GateMARMUX	write memory address register mux output onto bus
PCMUX	select value to write to program counter (2 bits)
DRMUX	select value to write to destination register (2 bits)
SR1MUX	select register to read from register file (2 bits)
ADDR1MUX	select register component of address (1 bit)
ADDR2MUX	select offset component of address (2 bits)
MARMUX	select type of address generation (1 bit)
ALUK	select arithmetic logic unit operation (2 bits)
MIO.EN	enable memory
R.W	read or write from memory

First, we have a set of seven 1-bit control signals (starting with "LD.") that specifies whether registers in the datapath load new values.

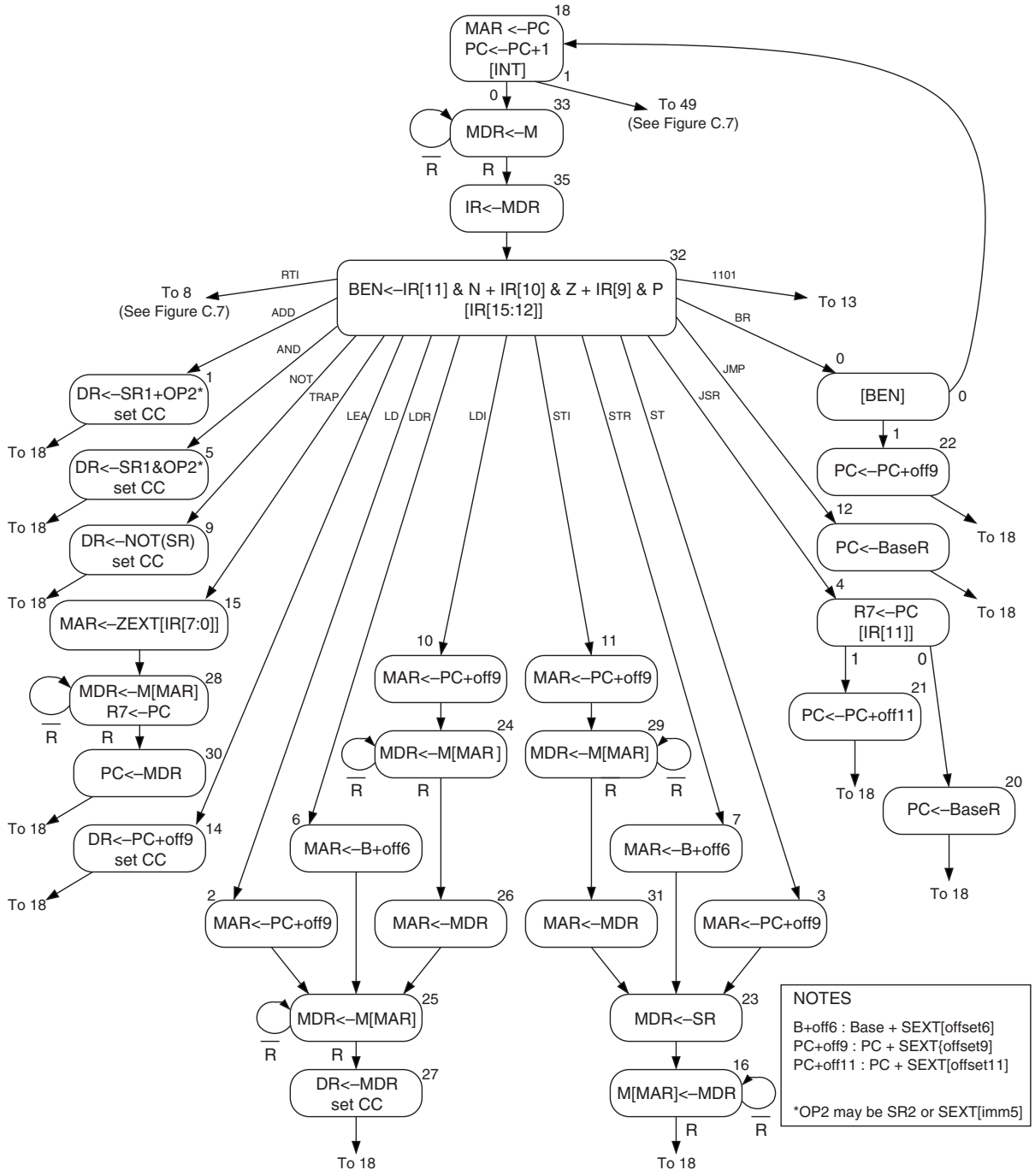
Next, there are four 1-bit signals (starting with "Gate") for tri-state buffers that control access to the bus. These four implement a distributed mux for the bus. Only one value can appear on the bus in any cycle, so at most one of these signals can be 1; others must all be 0 to avoid creating a short.



The third group (ending with "MUX") of signals controls multiplexers in the datapath. The number of bits for each depends on the number of inputs to the mux; the total number of signals is 10. The last two groups of signals control the ALU and the memory, requiring a total of 4 more signals. The total of all groups is thus 25 control signals for the datapath without support for privilege and interrupts.

4.1.2 Example Control Word: ADD

Before we begin to discuss control unit design in more detail, let's work through a couple of examples of implementing specific RTL with the control signals available. The figure below (Patt and Patel Figure C.2) shows a state machine for the LC-3 ISA (again without detail on interrupts and privilege).

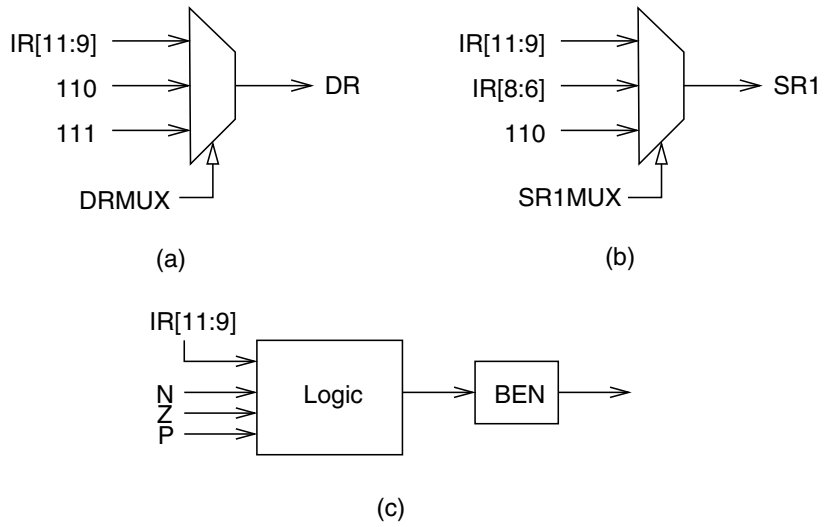


Consider now the state that implements the ADD instruction—state number 1 in the figure on the previous page, just below and to the left of the decode state. The RTL for the state is: DR ← SR + OP2, set CC.

We can think of the 25 control signals that implement the desired RTL as a **control word** for the datapath. Let's begin with the register load control signals. The RTL writes to two types of registers: the register file and the condition codes. To accomplish these simultaneous writes, LD.REG and LD.CC must be high. No other registers in the datapath should change, so the other five LD signals—LD.MAR, LD.MDR, LD.IR, LD.BEN, and LD.PC—should be low.

What about the bus? In order to write the result of the add operation into the register file, the control signals must allow the ALU to write its result on to the bus. So we need GateALU=1. And the other Gate signals—GatePC, GateMDR, and GateMARMUX—must all be 0. The condition codes are also calculated from the value on the bus, but they are calculated on the same value as is written to the register file (by the definition of the LC-3 ISA). If the RTL for an FSM state implicitly requires more than one value to appear on the bus in the same cycle, that state is impossible to implement using the given datapath. Either the datapath or the state machine must be changed in such a case. The textbook's design has been fairly thoroughly tested and debugged.

The earlier figure of the datapath does not show all of the muxes. The remaining muxes appear in the figure to the right (Patt and Patel Figure C.6).



Some of the muxes in the datapath must be used to enable the addition needed for ADD to occur. The DRMUX must select its IR[11:9] input in order to write to the destination register specified by the ADD instruction. Similarly, the SR1MUX must select its IR[8:6] input in order to pass the first source register specified by the ADD to the ALU as input A (see the datapath figure). SR2MUX is always controlled by the mode bit IR[5], so the control unit does not need to generate anything (note that this signal was not in the list given earlier). The rest of the muxes in the datapath—PCMUX, ADDR1MUX, ADDR2MUX, and MARMUX—do not matter, and the signals controlling them are don't cares. For example, since the PC does not change, the output of the PCMUX is simply discarded, thus which input the PCMUX forwards to its output cannot matter.

The ALU must perform an addition, so we must set the operation type ALUK appropriately. And memory should not be enabled (MIO.EN=0), in which case the read/write control for memory, R.W, is a don't care. These 25 signal values together (including seven don't cares) implement the RTL for the single state of execution for an ADD instruction.

4.1.3 Example Control Word: LDR

As a second example, consider the first state in the sequence that implements the LDR instruction—state number 6 in the figure on the previous page. The RTL for the state is: MAR ← BaseR + off6, but BaseR is abbreviated to "B" in the state diagram.

What is the control word for this state? Let's again begin with the register load control signals. Only the MAR is written by the RTL, so we need LD.MAR=1 and the other load signals all equal to 0.

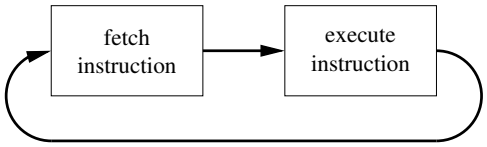
The address (BaseR + off6) is generated by the address adder, then passes through the MARMUX to the bus, from which it can be written into the MAR. To allow the MARMUX to write to the bus, we set GateMARMUX high and set the other three Gate control signals low.

More of the muxes are needed for this state's RTL than we needed for the ADD execution state's RTL. The SR1MUX must again select its IR[8:6] input, this time in order to pass the BaseR specified by the instruction to ADDR1MUX. ADDR1MUX must then select the output of the register file in order to pass the BaseR to the address adder. The other input of the address adder should be off6, which corresponds to the sign-extended version of IR[5:0]. ADDR2MUX must select this input to pass to the address adder. Finally, the MARMUX must select the output of the address adder. The PCMUX and DRMUX do not matter for this case, and can be left as don't cares. Neither the PC nor any register in the register file is written.

The output of the ALU is not used, so which operation it performs is irrelevant, and the ALUK controls are also don't cares. Memory is also not used, so again we set MIO.EN=0. And, as before, the R.W control for memory is a don't care. These 25 signal values together (again including seven don't cares) implement the RTL for the first state of LDR execution.

4.1.4 Hardwired Control

Now we are ready to think about how control signals can be generated. As illustrated to the right, instruction processing consists of two steps repeated infinitely: fetch an instruction, then execute the instruction.



Let's say that we choose a fixed number of cycles for each of these two steps. We can then control our system with a counter, using the counter's value and the IR to generate the control signals through combinational logic. The PC is used only as data and has little or no direct effect on how the system fetches and processes instructions.[15] This approach in general is called **hardwired control**.

How many cycles do we need for instruction fetch? How many cycles do we need for instruction processing? The answers depend on the factors: the complexity of the ISA, and the complexity of the datapath.

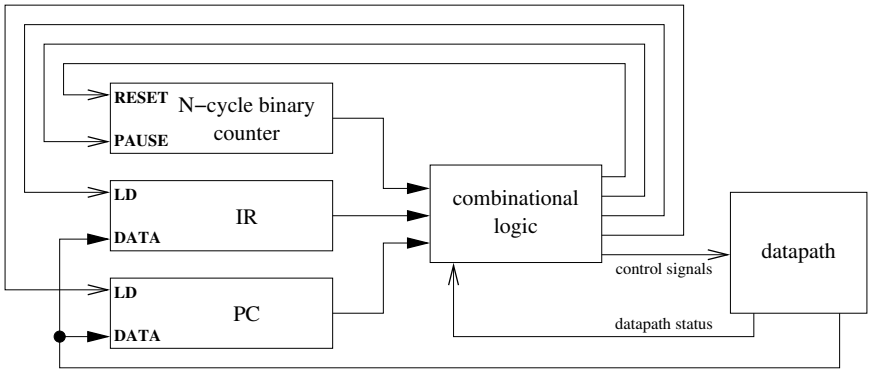
Given a simple ISA, we can design a datapath that is powerful enough to process any instruction in a single cycle. The control unit for such a design is an example of **single-cycle, hardwired control**. While this approach simplifies the control unit, the cycle time for the clock is limited by the slowest instruction. The clock must also be slow enough to let memory operations complete in single cycle, both for instruction fetch and for instructions that require a memory access. Such a low clock rate is usually not acceptable.

More generally, we can use a simpler datapath and break both instruction fetch and instruction processing into multiple steps. Using the datapath in the figure from Patt and Patel, for example, instruction fetch requires three steps, and the number of steps for instruction processing depends on the type of instruction being processed. The state diagram shown earlier illustrates the steps for each opcode.

Although the datapath is not powerful enough to complete instructions in a single cycle, we can build a **multi-cycle, hardwired control** unit (one based on combinational logic). In fact, this type of control unit is not much more complex than the single-cycle version. For the control unit's FSM, we can use a binary counter to enumerate first the steps of fetch, then the steps of processing. The counter value along with the IR register can drive combinational logic to produce control signals. And, to avoid processing at the speed of the slowest instruction (the opcode that requires the largest number of steps; LDI and STI in the LC-3 state diagram), we can add a reset signal to the counter to force it back to instruction fetch. The FSM counter reset signal is simply another control signal. Finally, we can add one more signal that pauses the counter while waiting for a memory operation to complete. The system clock can then run at the speed of the logic rather than at the speed of memory. The control unit implementation discussed in Appendix C of Patt and Patel is not hardwired, but it does make use of a memory ready signal to achieve this decoupling between the clock speed of the processor and the access time of the memory.

[15] Some ISAs do split the address space into privileged and non-privileged regions, but we ignore that possibility here.

The figure to the right illustrates a general multi-cycle, hardwired control unit. The three blocks on the left are the control unit state. The combinational logic in the middle uses the control unit state along with some datapath status bits to compute the control signals for the datapath and the extra controls needed for the FSM counter, IR, and PC. The datapath appears to the right in the figure.



How complex is the combinational logic? As mentioned earlier, we assume that the PC does not directly affect control. But we still have 16 bits of IR, the FSM counter state, and the datapath status signals. Perhaps we need 24-variable K-maps? Here's where engineering and human design come to the rescue: by careful design of the ISA and the encoding, the authors have made many of the datapath control signals for the LC-3 ISA quite simple. For example, the register that appears on the register file's SR2 output is always specified by IR[2:0]. The SR1 output requires a mux, but the choices are limited to IR[11:9] and IR[8:6] (and R6 in the design with support for interrupts). Similarly, the destination register in the register file is always R7 or IR[11:9] (or, again, R6 when supporting interrupts). The control signals for an LC-3 datapath depend almost entirely on the state of the control unit FSM (counter bits in a hardwired design) and the opcode IR[15:12]. The control signals are thus reduced to fairly simple functions.

Let's imagine building a hardwired control unit for the LC-3. Let's start by being more precise about the number of inputs to the combinational logic. Although most decisions are based on the opcode, the datapath and state diagram shown earlier for the LC-3 ISA do have one instance of using another instruction bit to determine behavior. Specifically, the JSR instruction has two modes, and the control unit uses IR[11] to choose between them. So we need to have five bits of IR instead of four as input to our logic.

How many datapath status signals are needed? When the control unit accesses memory, it must wait until the memory finishes the access, as indicated by a memory ready signal R. And the control unit must implement the conditional part of conditional branches, for which it uses the datapath's branch enable signal BEN. These two datapath status signals suffice for our design.

How many bits do we need for the counter? Instruction fetch requires three cycles: one to move the PC to the MAR and increment the PC, a second to read from memory into MDR, and a third to move the instruction bits across the bus from MDR into IR. Instruction decoding in a hardwired design is implicit and requires no cycles: since all of our control signals can depend on the IR, we do not need a cycle to change the FSM state to reflect the opcode. Looking at the LC-3 state diagram, we see that processing an instruction requires at most five cycles. In total, at most eight steps are needed to fetch and process any LC-3 instruction, so we can use a 3-bit binary counter.

We thus have a total of ten bits of input: IR[15:11], R, BEN, and a 3-bit counter. Adding the RESET and PAUSE controls for our FSM counter to the 25 control signals listed earlier, we need to find 27 functions on 10 variables. That's still a lot of big K-maps to solve. Is there an easier way?

4.1.5 Using a Memory for Logic Functions

Consider a case in which you need to compute many functions on a small number of bits, such as we just described for the multi-cycle, hardwired control unit. One strategy is to use a memory (possibly a read-only memory). A $2^m \times N$ memory can be viewed as computing N arbitrary functions on m variables. The functions to be computed are specified by filling in the bits of the memory. So long as the value of m is fairly small, the memory (especially SRAM) can be fast.

Synthesis tools (or hard work) can, of course, produce smaller designs that use fewer gates. Actually, tools may be able to optimize a fixed design expressed as read-only memory, too. But designing the functions with a memory makes them easier to modify later. If we make a mistake, for example, in computing one of the functions, we need only change a bit or two in the memory instead of solving equations and reoptimizing and replacing logic gates. We can also extend our design if we have space remaining (that is, if the functions are undefined for some combinations of the m inputs). The Cray T3D supercomputer, for example, used a similar approach to add new instructions to the Alpha processors on which it was based.

This strategy is effective in many contexts, so let's briefly discuss two analogous cases. In software, a memory becomes a lookup table. Before handheld calculators, lookup tables were used by humans to compute transcendental functions such as sines, cosines, logarithms. Computer graphics hardware and software used a similar approach for transcendental functions in order to reduce cost and improve speed. Functions such as counting 1 bits in a word are useful for processor scheduling and networking, but not all ISAs provide this type of instruction. In such cases, lookup tables in software are often the best solution.

In programmable hardware such as Field Programmable Gate Arrays (FPGAs), lookup tables (called LUTs in this context) have played an important role in implementing arbitrary logic functions. The FPGA is the modern form of the programmable logic array (PLA) mentioned in the textbook, and will be your main tool for developing digital hardware in ECE385. For many years, FPGAs served as a hardware prototyping platform, but many companies today ship their first round products using designs mapped to FPGAs. Why? Chips are more and more expensive to design, and mistakes are costly to fix. In contrast, while companies pay more to buy an FPGA than to produce a chip (after the first chip!), errors in the design can usually be fixed by sending customers a new version through the Internet.

Let's return to our LC-3 example. Instead of solving the K-maps, we can use a small memory: $2^{10} \times 27$ bits (27,648 bits total). We just need calculate the bits, put them into the memory, and use the memory to produce the control signals. The "address" input to the memory are the same 10 bits that we needed for our combinational logic: IR[15:11], R, BEN, and the FSM counter. The data outputs of the memory are the control signals and the RESET and PAUSE inputs to the FSM counter. And we're done.

We can do a little better, though. The datapath in the textbook was designed to work with the textbook's control unit. If we add a little logic, we can significantly simplify our memory-based, hardwired implementation. For example, we only need to pause the FSM counter when waiting for memory. If we can produce a control signal that indicates a need to wait for memory, say WAIT-MEM, we can use a couple of gates to compute the FSM counter's PAUSE signal as WAIT-MEM AND (NOT R). Making this change shrinks our memory to $2^9 \times 27$ bits. The extra two control signals in this case are RESET and WAIT-MEM.

Next, look at how BEN is used in the state diagram: the only use is to terminate the processing of branch instructions when no branch should occur (when BEN=0). We can fold that functionality into the FSM counter's RESET signal by producing a branch reset signal, BR-RESET, to reset the counter to end a branch and a second signal, INST-DONE, when an instruction is done. The RESET input for the FSM counter is then (BR-RESET AND (NOT BEN)) OR INST-DONE. And our memory further shrinks to $2^8 \times 28$ bits, where the extra three control signals are WAIT-MEM, BR-RESET, and INST-DONE.

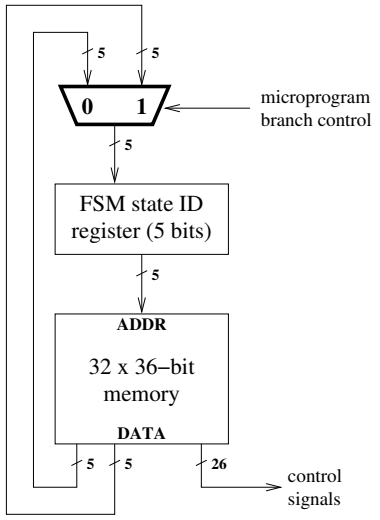
Finally, recall that the only need for IR[11] is to implement the two forms of JSR. But we can add wires to connect SR1 to PCMUX's fourth input, then control the PCMUX output selection using IR[11] when appropriate (using another control signal). With this extension, we can implement both forms with a single state, writing to both R7 and PC in the same cycle. Our final memory can then be $2^7 \times 29$ bits (3,712 bits total), which is less than one-seventh the number of bits that we needed before modifying the datapath.

4.1.6 Microprogrammed Control

We are now ready to discuss the second approach to control unit design. Take another look at the state diagram for the the LC-3 ISA. Does it remind you of anything? Like a flowchart, it has relatively few arcs leaving each state—usually only one or two.

What if we treat the state diagram as a program? We can use a small memory to hold **microinstructions** (another name for control words) and use the FSM state number as the memory address. Without support for interrupts or privilege, and with the datapath extension for JSR mentioned for hardwired control, the LC-3 state machine requires fewer than 32 states. The datapath has 25 control signals, but we need one more for the datapath extension for JSR. We thus start with 5-bit state number (in a register) and a $2^5 \times 26$ bit memory, which we call our control ROM (read-only memory) to distinguish it from the big, slow, von Neumann memory. Each cycle, the **microprogrammed control** unit applies the FSM state number to the control ROM (no IR bits, just the state number), gets back a set of control signals, and uses them to drive the datapath.

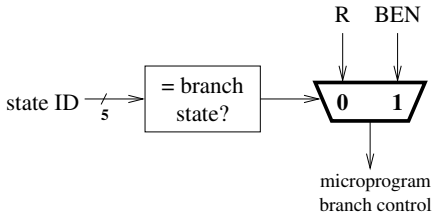
To write our microprogram, we need to calculate the control signals for each microinstruction and put them in the control ROM, but we also need to have a way to decide which microinstruction should execute next. We call the latter problem **sequencing** or microsequencing.



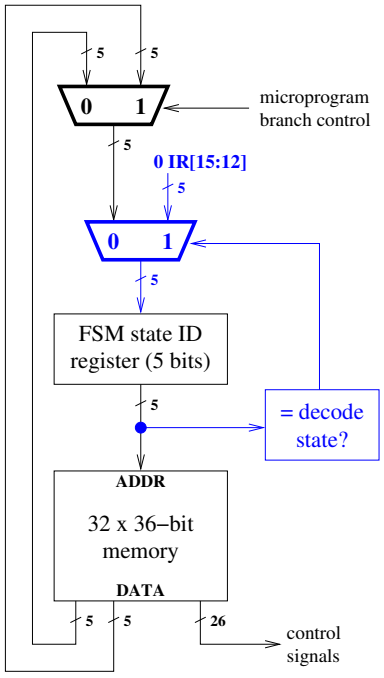
Notice that most of the time there's no choice: we have only *one* next microinstruction. One simple approach is then to add the address (the 5-bit state ID) of the next microinstruction to the control ROM. Instead of 26 bits per FSM state, we now have 31 bits per FSM state.

Sometimes we do need to have two possible next states. When waiting for memory (the von Neumann memory, not the control ROM) to finish an access, for example, we want our FSM to stay in the same state, then move to the next state when the access completes. Let's add a second address to each microinstruction, and add a branch control signal for the microprogram to decide whether we should use the first address or the second for the next microinstruction. This design, using a $2^5 \times 36$ bit memory (1,152 bits total), appears to the right.

The microprogram branch control signal is a Boolean logic expression based on the memory ready signal R and IR[11]. We can implement it with a state ID comparison and a mux, as shown to the right. For the branch instruction execution state, the mux selects input 1, BEN. For all other states, the mux selects input 0, R. When an FSM state has only a single next state, we set both IDs in the control ROM to the ID for that state, so the value of R has no effect.



What's missing? Decode! We do have one FSM state in which we need to be able to branch to one of sixteen possible next states, one for each opcode. Let's just add another mux and choose the state IDs for starting to process each opcode in an easy way, as shown to the right with extensions highlighted in blue. The textbook assigns the first state for processing each opcode the IDs IR[15:12] preceeded by two 0s (the textbook's design requires 6-bit state IDs). We adopt the same strategy. For example, the first FSM state for processing an ADD is 00001, and the first state for a TRAP is 01111. In each case, the opcode encoding specifies the last four bits.



Now we can pick the remaining state IDs arbitrarily and fill in the control ROM with control signals that implement each state's RTL and the two possible next states. Transitions from the decode state are handled by the extra mux.

The microprogrammed control unit implementation in Appendix C of Patt and Patel is similar to the one that we have developed here, but is slightly more complex so that it can handle interrupts and privilege.