

6

THE CONTROL UNIT

In the earlier chapters, we examined the computer at the Application Level, the High Level Language level, and the Assembly Language level (as shown in Figure 1-4.) In Chapter 4 we introduced the concept of an ISA: an instruction set that effects operations on registers and memory. In this chapter, we explore the part of the machine that is responsible for implementing these operations: the control unit of the CPU. In this context, we view the machine at the **microarchitecture** level (the Microprogrammed/Hardwired Control level in Figure 1-4.) The microarchitecture consists of the control unit and the programmer-visible registers, functional units such as the ALU, and any additional registers that may be required by the control unit.

A given ISA may be implemented with different microarchitectures. For example, the Intel Pentium ISA has been implemented in different ways, all of which support the same ISA. Not only Intel, but a number of competitors such as AMD and Cyrix have implemented Pentium ISAs. A certain microarchitecture might stress high instruction execution speed, while another stresses low power consumption, and another, low processor cost. Being able to modify the microarchitecture while keeping the ISA unchanged means that processor vendors can take advantage of new IC and memory technology while affording the user upward compatibility for their software investment. Programs run unchanged on different processors as long as the processors implement the same ISA, regardless of the underlying microarchitectures.

In this chapter we examine two polarizingly different microarchitecture approaches: microprogrammed control units and hardwired control units, and we examine them by showing how a subset of the ARC processor can be implemented using these two design techniques.

6.1 Basics of the Microarchitecture

The functionality of the microarchitecture centers around the fetch-execute cycle, which is in some sense the “heart” of the machine. As discussed in Chapter 4, the steps involved in the fetch-execute cycle are:

- 1) Fetch the next instruction to be executed from memory.
- 2) Decode the opcode.
- 3) Read operand(s) from main memory, if any.
- 4) Execute the instruction and store results.
- 5) Go to Step 1.

It is the microarchitecture that is responsible for making these five steps happen. The microarchitecture fetches the next instruction to be executed, determines which instruction it is, fetches the operands, executes the instruction, stores the results, and then repeats.

The microarchitecture consists of a **data section** which contains registers and an ALU, and a **control section**, as illustrated in Figure 6-1. The data section is also

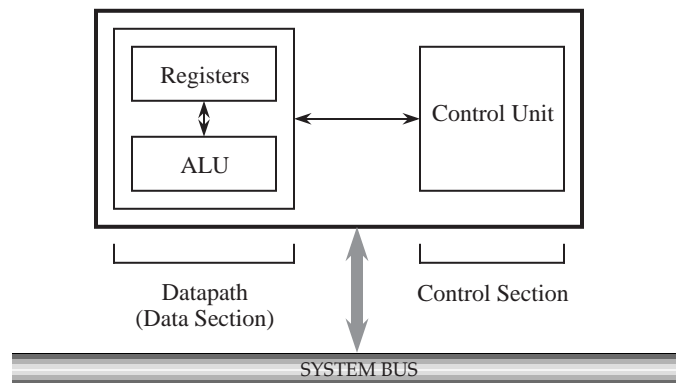


Figure 6-1 High level view of a microarchitecture.

referred to as the **datapath**. Microprogrammed control uses a special purpose **microprogram**, not visible to the user, to implement operations on the registers and on other parts of the machine. Often, the microprogram contains many program steps that collectively implement a single (macro)instruction. **Hardwired**

control units adopt the view that the steps to be taken to implement an operation comprise states in a finite state machine, and the design proceeds using conventional digital design methods (such as the methods covered in Appendix A.) In either case, the datapath remains largely unchanged, although there may be minor differences to support the differing forms of control. In designing the ARC control unit, the microprogrammed approach will be explored first, and then the hardwired approach, and for both cases the datapath will remain unchanged.

6.2 A Microarchitecture for the ARC

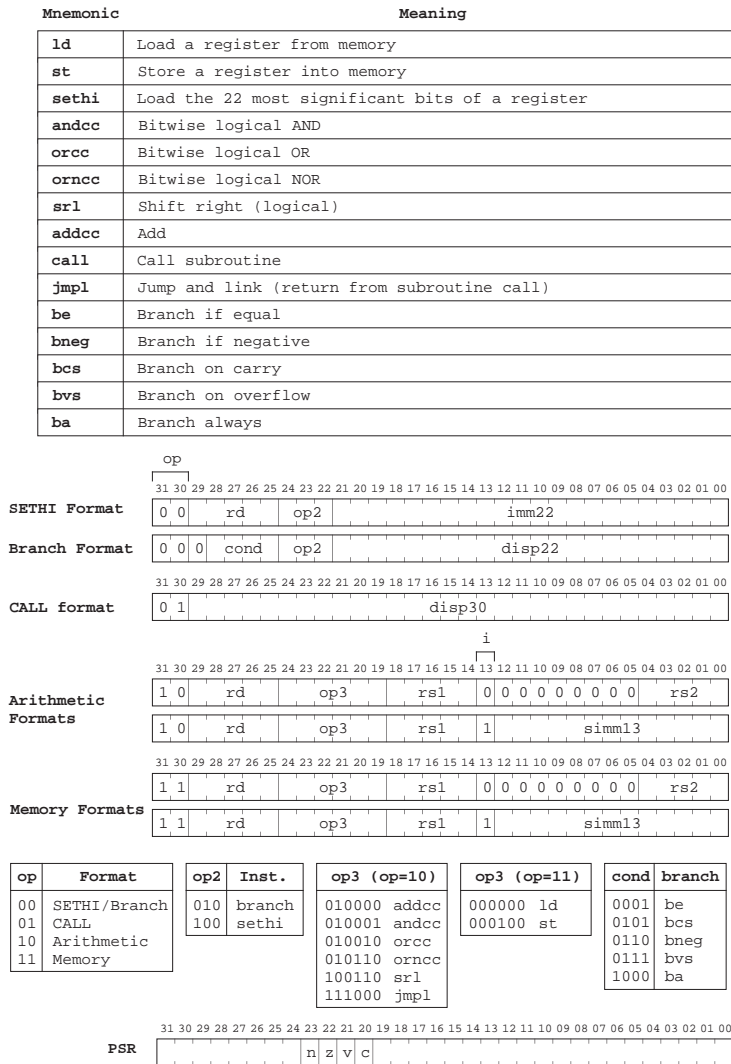
In this section we consider a microprogrammed approach for designing the ARC control unit. We begin by describing the datapath and its associated control signals.

The instruction set and instruction format for the ARC subset is repeated from Chapter 4 in Figure 6-2. There are 15 instructions that are grouped into four formats according to the leftmost two bits of the coded instruction. The Processor Status Register `%psr` is also shown.

6.2.1 THE DATAPATH

A datapath for the ARC is illustrated in Figure 6-3. The datapath contains 32 user-visible data registers (`%r0` – `%r31`), the program counter (`%pc`), the instruction register (`%ir`), the ALU, four temporary registers not visible at the ISA level (`%temp0` – `%temp3`), and the connections among these components. The number adjacent to a diagonal slash on some of the lines is a simplification that indicates the number of separate wires that are represented by the corresponding single line.

Registers `%r0` – `%r31` are directly accessible by a user. Register `%r0` always contains the value 0, and cannot be changed. The `%pc` register is the program counter, which keeps track of the next instruction to be read from the main memory. The user has direct access to `%pc` only through the `call` and `jmp1` instructions. The temporary registers are used in interpreting the ARC instruction set, and are not visible to the user. The `%ir` register holds the current instruction that is being executed. It is not visible to the user.



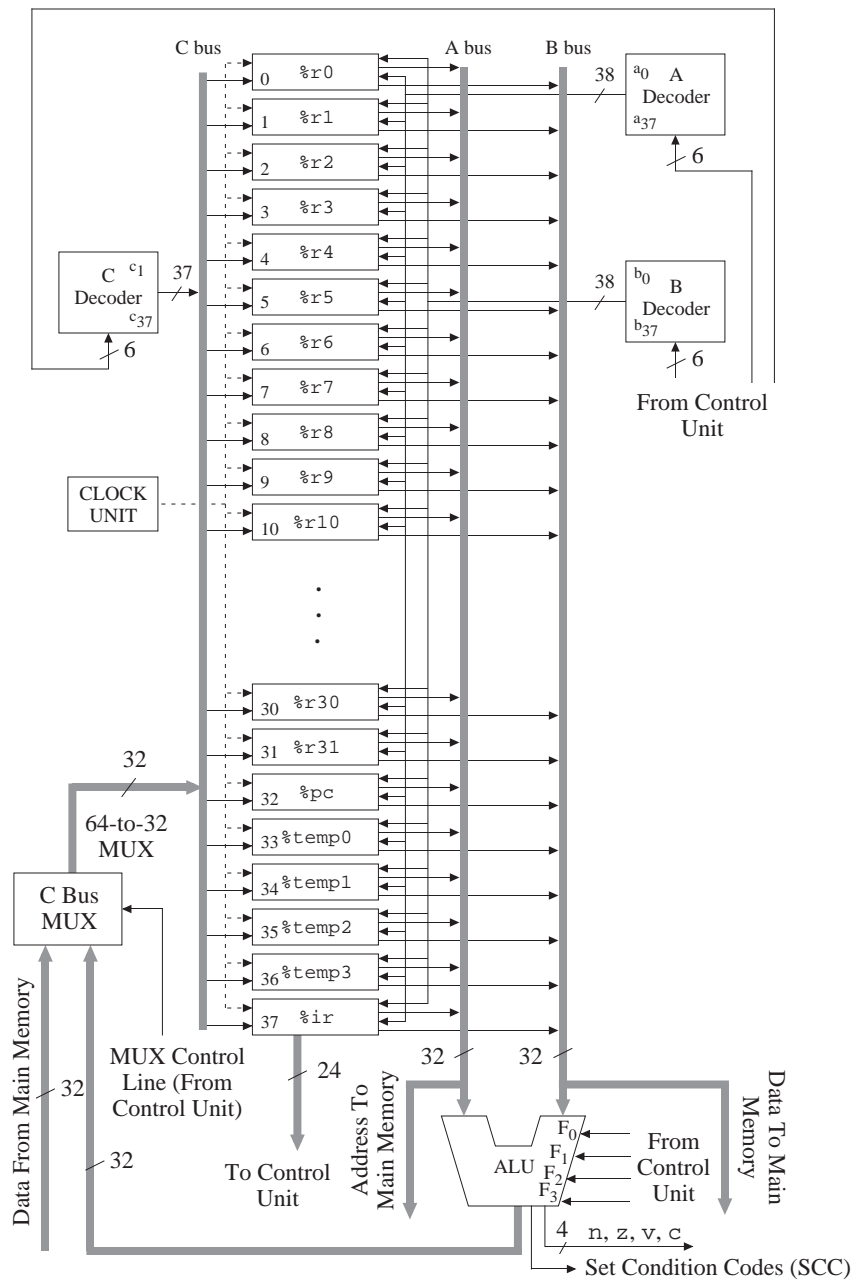


Figure 6-3 The datapath of the ARC.

The ANDCC and AND operations perform a bit-by-bit logical AND of corresponding bits on the A and B busses. Note that only operations that end with

F_3 F_2 F_1 F_0	Operation	Changes Condition Codes
0 0 0 0	ANDCC (A, B)	yes
0 0 0 1	ORCC (A, B)	yes
0 0 1 0	NORCC (A, B)	yes
0 0 1 1	ADDCC (A, B)	yes
0 1 0 0	SRL (A, B)	no
0 1 0 1	AND (A, B)	no
0 1 1 0	OR (A, B)	no
0 1 1 1	NOR (A, B)	no
1 0 0 0	ADD (A, B)	no
1 0 0 1	LSHIFT2 (A)	no
1 0 1 0	LSHIFT10 (A)	no
1 0 1 1	SIMM13 (A)	no
1 1 0 0	SEXT13 (A)	no
1 1 0 1	INC (A)	no
1 1 1 0	INCPC (A)	no
1 1 1 1	RSHIFT5 (A)	no

Figure 6-4 ARC ALU operations.

“CC” affect the condition codes, and so ANDCC affects the condition codes whereas AND does not. (There are times when we wish to execute arithmetic and logic instructions without disturbing the condition codes.) The ORCC and OR operations perform a bit-by-bit logical OR of corresponding bits on the A and B busses. The NORCC and NOR operations perform a bit-by-bit logical NOR of corresponding bits on the A and B busses. The ADDCC and ADD operations carry out addition using two’s complement arithmetic on the A and B busses.

The SRL (shift right logical) operation shifts the contents of the A bus to the right by the amount specified on the B bus (from 0 to 31 bits). Zeros are copied into the leftmost bits of the shifted result, and the rightmost bits of the result are discarded. LSHIFT2 and LSHIFT10 shift the contents of the A bus to the left by two and 10 bits, respectively. Zeros are copied into the rightmost bits.

SIMM13 retrieves the least significant 13 bits of the A bus, and places zeros in the 19 most significant bits. SEXT13 performs a sign extension of the 13 least significant bits on the A bus to form a 32-bit word. That is, if the leftmost bit of the 13 bit group is 1, then 1’s are copied into the 19 most significant bits of the result, otherwise, 0’s are copied into the 19 most significant bits of the result. The INC operation increments the value on the A bus by 1, and the INCPC operation increments the value on the A bus by four, which is used in incrementing

the PC register by one word (four bytes). `INCPC` can be used on any register placed on the A bus.

The `RSHIFT5` operation shifts the operand on the A bus to the right by 5 bits, copying the leftmost bit (the sign bit) into the 5 new bits on the left. This has the effect of performing a 5-bit sign extension. When applied three times in succession to a 32-bit instruction, this operation also has the effect of placing the leftmost bit of the `COND` field in the Branch format (refer to Figure 6-2) into the position of bit 13. This operation is useful in decoding the Branch instructions, as we will see later in the chapter. The sign extension for this case is inconsequential.

Every arithmetic and logic operation can be implemented with just these ALU operations. As an example, a subtraction operation can be implemented by forming the two's complement negative of the subtrahend (making use of the `NOR` operation and adding 1 to it with `INC`) and then performing addition on the operands. A shift to the left by one bit can be performed by adding a number to itself. A “do-nothing” operation, which is frequently needed for simply passing data through the ALU without changing it, can be implemented by logically ANDing an operand with itself and discarding the result in `%r0`. A logical XOR can be implemented with the `AND`, `OR`, and `NOR` operations, making use of DeMorgan's theorem (see problem 6.5).

The ALU generates the `c`, `n`, `z`, and `v` condition codes which are true for a carry, negative, zero, or overflow result, respectively. The condition codes are changed only for the operations indicated in Figure 6-4. A signal (SCC) is also generated that tells the `%psr` register when to update the condition codes.

The ALU can be implemented in a number of ways. For the sake of simplicity, let us consider using a **lookup table** (LUT) approach. The ALU has two 32-bit data inputs *A* and *B*, a 32-bit data output *C*, a four-bit control input *F*, a four-bit condition code output (*N*, *V*, *C*, *Z*), and a signal (SCC) that sets the flags in the `%psr` register. We can decompose the ALU into a cascade of 32 LUTs that implement the arithmetic and logic functions, followed by a **barrel shifter** that implements the shifts. A block diagram is shown in Figure 6-5.

The barrel shifter shifts the input word by an arbitrary amount (from 0 to 31 bits) according to the settings of the control inputs. The barrel shifter performs shifts in levels, in which a different bit of the Shift Amount (SA) input is observed at each level. A partial gate-level layout for the barrel shifter is shown in

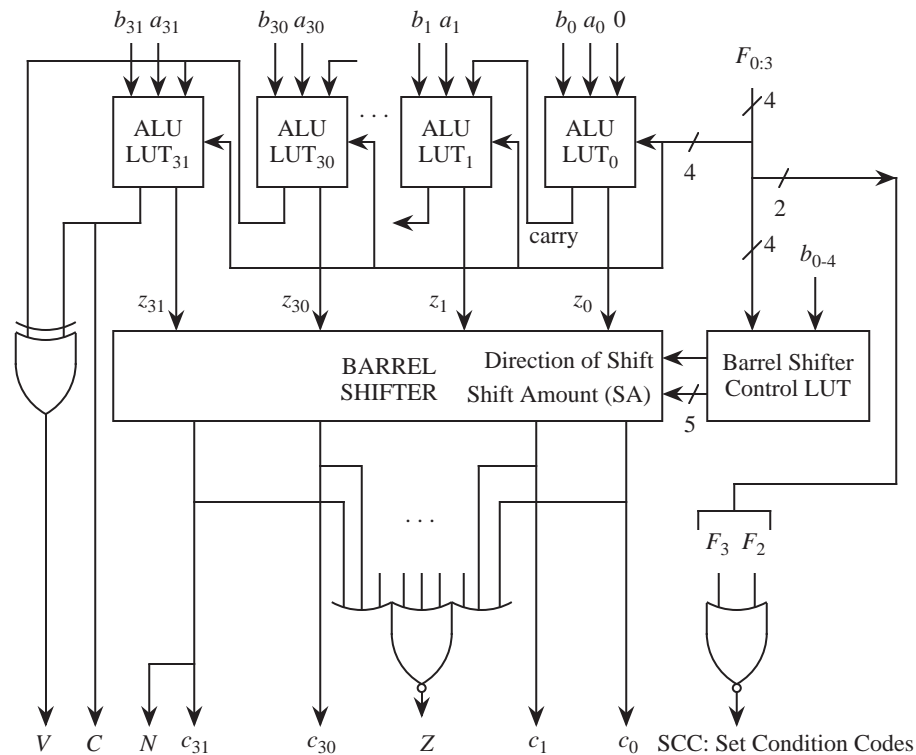


Figure 6-5 Block diagram of the 32-bit ALU.

Figure 6-6. Starting at the bottom of the circuit, we can see that the outputs of the bottom stage will be the same as the inputs to that stage if the SA_0 bit is 0. If the SA_0 bit is 1, then each output position will take on the value of its immediate left or right neighbor, according to the direction of the shift, which is indicated by the Shift Right input. At the next higher level, the method is applied again, except that the SA_1 bit is observed and the amount of the shift is doubled. The process continues until bit SA_4 is observed at the highest level. Zeros are copied into positions that have no corresponding inputs. With this structure, an arbitrary shift from 0 to 31 bits to the left or the right can be implemented.

Each of the 32 ALU LUTs is implemented (almost) identically, using the same lookup table entries, except for changes in certain positions such as for the `INC` and `INCPC` operations (see problem Figure 6.20). The first few entries for each LUT are shown in Figure 6-7. The barrel shifter control LUT is constructed in a similar manner, but with different LUT entries.

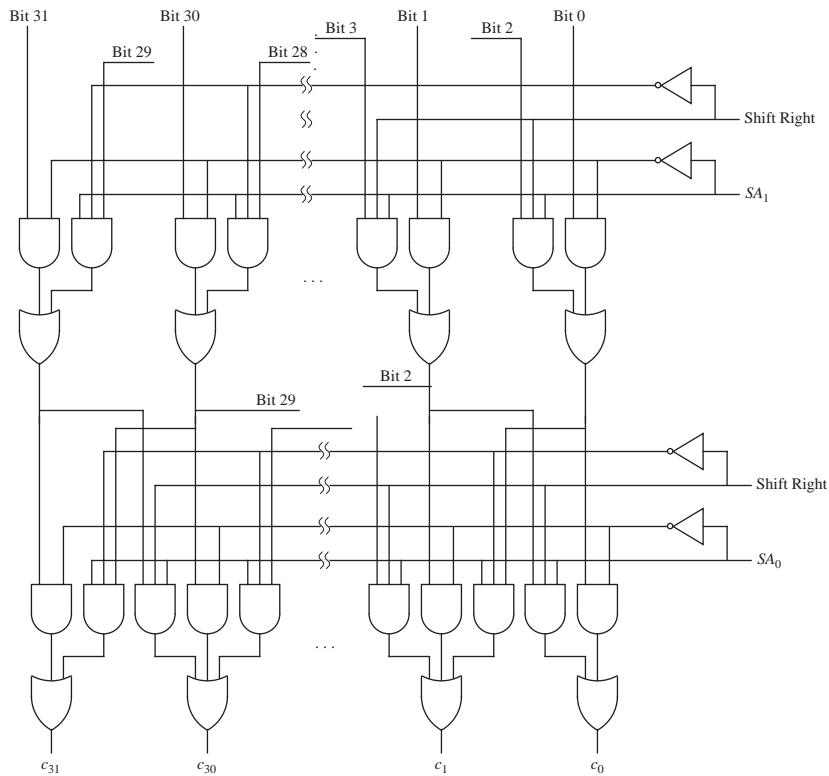


Figure 6-6 Gate-level layout of barrel shifter.

The condition code bits n , z , v , and c are implemented directly. The n and c bits are taken directly from the c_{31} output of the barrel shifter and the carry-out position of ALU LUT₃₁, respectively. The z bit is computed as the NOR over the barrel shifter outputs. The z bit is 1 only if all of the barrel shifter outputs are 0. The v (overflow) bit is set if the carry into the most significant position is different than the carry out of the most significant position, which is implemented with an XOR gate.

Only the operations that end in “CC” should set the condition codes, and so a signal is generated that informs the condition codes to change, as indicated by the label “SCC: Set Condition Codes.” This signal is true when both F_3 and F_2 are false.

The Registers

All of the registers are composed of *falling edge-triggered* D flip-flops (see Appen-

	F_3	F_2	F_1	F_0	Carry In	a_i	b_i	z_i	Carry Out
ANDCC	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	1	1	1	0
	0	0	0	0	1	0	0	0	0
	0	0	0	0	1	0	1	0	0
	0	0	0	0	1	1	0	0	0
	0	0	0	0	1	1	1	1	0
ORCC	0	0	0	1	0	0	0	0	0
	0	0	0	1	0	0	1	1	0
	0	0	0	1	0	1	0	1	0
	0	0	0	1	0	1	1	1	0
	0	0	0	1	1	0	0	0	0
	0	0	0	1	1	0	1	1	0
	0	0	0	1	1	0	1	1	0
	0	0	0	1	1	0	1	1	0
					.			.	.
					.			.	.
					.			.	.

Figure 6-7 Truth table for most of the ALU LUTs.

dix A). This means that the outputs of the flip-flops do not change until the clock makes a transition from high to low (the *falling edge* of the clock). The registers all take a similar form, and so we will only look at the design of register %r1. All of the datapath registers are 32 bits wide, and so 32 flip-flops are used for the design of %r1, which is illustrated in Figure 6-8.

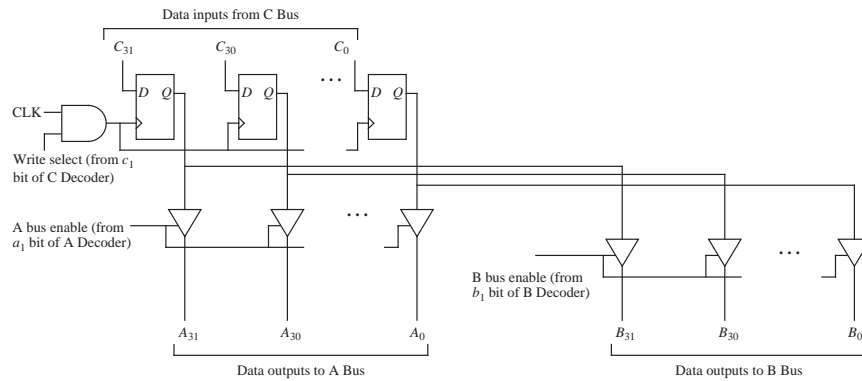


Figure 6-8 Design of register %r1.

The CLK input to register %r1 is ANDed with the select line (c_1) from the C Decoder. This ensures that %r1 only changes when the control section instructs it to change. The data inputs to %r1 are taken directly from the corresponding

lines of the C bus. The outputs are written to the corresponding lines of the A and B busses through tri-state buffers, which are “electrically disconnected” unless their enable inputs are set to 1. The outputs of the buffers are enabled onto the A and B busses by the a_1 and b_1 outputs of the A and B decoders, respectively. If neither a_1 nor b_1 are high (meaning they are equal to 1), then the outputs of $\%r1$ are electrically disconnected from both the A and B busses since the tri-state buffers are disabled.

The remaining registers take a similar form, with a few exceptions. Register $\%r0$ always contains a 0, which cannot be changed. Register $\%r0$ thus has no inputs from the C bus nor any inputs from the C decoder, and does not need flip-flops (see Problem 6.11). The $\%ir$ register has additional outputs that correspond to the rd, rs1, rs2, op, op2, op3, and bit 13 fields of an instruction, as illustrated in Figure 6-9. These outputs are used by the control section in interpreting

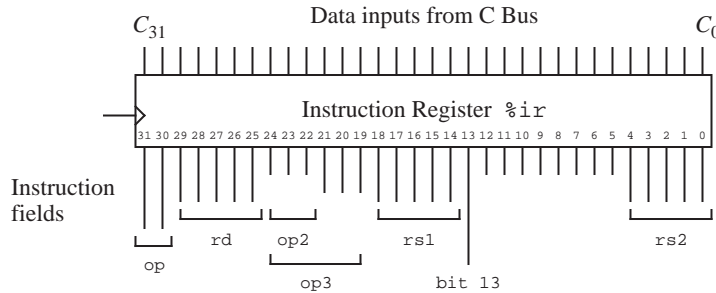


Figure 6-9 Outputs to control unit from register $\%ir$.

an instruction as we will see in Section 6.2.4.

The A, B, and C decoders shown in Figure 6-3 simplify register selection. The six-bit inputs to the decoders select a single register for each of the A, B, and C busses. There are $2^6 = 64$ possible outputs from the decoders, but there are only 38 data registers. The index shown to the left of each register (in base 10) in Figure 6-3 indicates the value that must be applied to a decoder input to select the corresponding register. The 0 output of the C decoder is not used because $\%r0$ cannot be written. Indices that are greater than 37 do not correspond to any registers, and are free to be used when no registers are to be connected to a bus.

6.2.2 THE CONTROL SECTION

The entire microprogrammed ARC microarchitecture is shown in Figure 6-10. The figure shows the datapath, the control unit, and the connections between

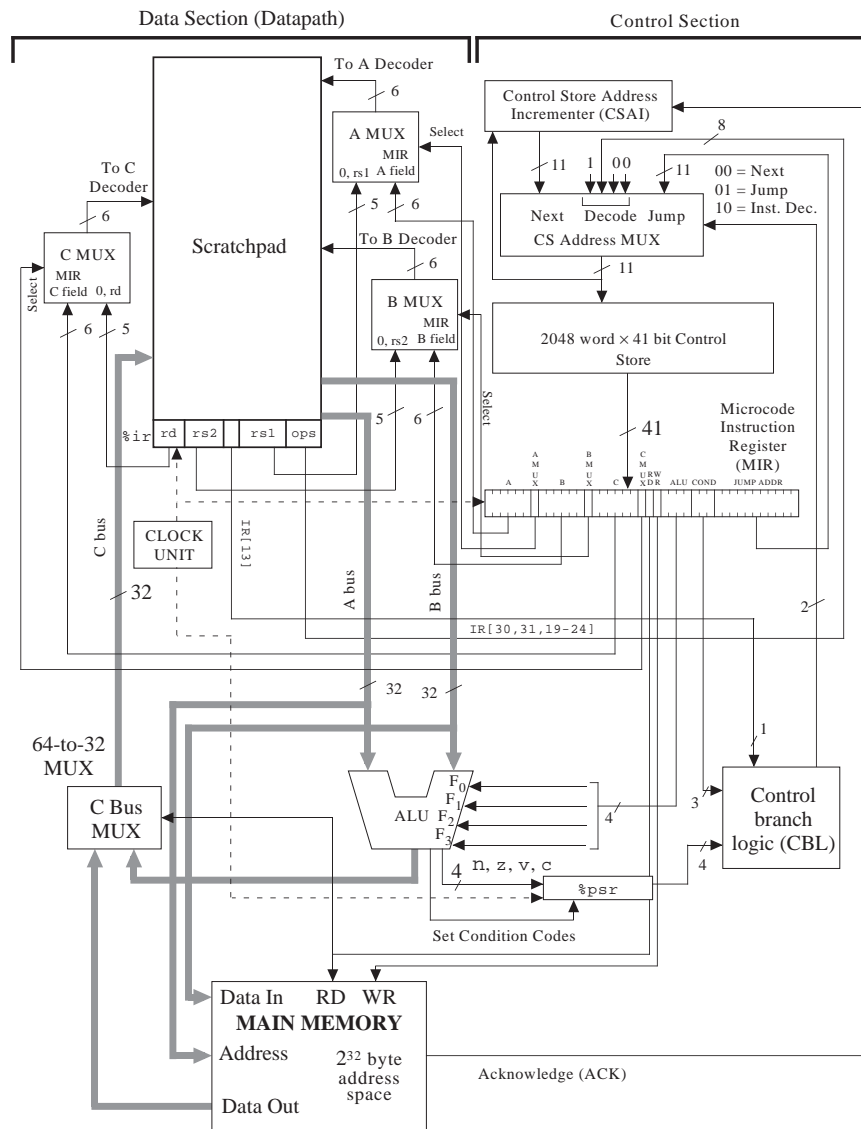


Figure 6-10 The microarchitecture of the ARC.

them. At the heart of the control unit is a 2048 word \times 41 bit read-only memory (ROM) that contains values for all of the lines that must be controlled to implement each user-level instruction. The ROM is referred to as a **control store** in this context. Each 41-bit word is called a **microinstruction**. The control unit is responsible for fetching microinstructions and executing them, much in the

same way as user-level ARC macroinstructions are fetched and executed. This microinstruction execution is controlled by the microprogram instruction register (MIR), the processor status register (`%psr`), and a mechanism for determining the next microinstruction to be executed: the Control Branch Logic (CBL) unit and the Control Store (CS) Address MUX. A separate PC for the microprogram is not needed to store the address of the next microinstruction, because it is recomputed on every clock cycle and therefore does not need to be stored for future cycles.

When the microarchitecture begins operation (at power-on time, for example), a reset circuit (not shown) places the microword at location 0 in the control store into the MIR and executes it. From that point onward, a microword is selected for execution from either the Next, the Decode, or the Jump inputs to the CS Address MUX, according to the settings in the COND field of the MIR and the output of the CBL logic. After each microword is placed in the MIR, the datapath performs operations according to the settings in the individual fields of the MIR. This process is detailed below.

A microword contains 41 bits that comprise 11 fields as shown in Figure 6-11.

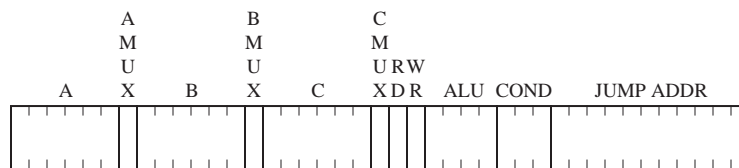


Figure 6-11 The microword format.

Starting from the left, the A field determines which of the registers in the datapath are to be placed on the A bus. The bit patterns for the registers correspond to the binary representations of the base 10 register indices shown in Figure 6-3 (000000 – 100101). The AMUX field selects whether the A Decoder takes its input from the A field of the MIR (AMUX = 0) or from the `rs1` field of `%ir` (AMUX = 1).

In a similar manner, the B field determines which of the registers in the datapath are to be placed on the B bus. The BMUX field selects whether the B Decoder takes its input from the B field of the MIR (BMUX = 0) or from the `rs2` field of `%ir` (BMUX = 1). The C field determines which of the registers in the datapath is to be written from the C bus. The CMUX field selects whether the C Decoder takes its input from the C field of the MIR (CMUX = 0) or from the `rd` field of

`%ir` (`CMUX = 1`). Since `%r0` cannot be changed, the bit pattern 000000 can be used in the C field when none of these registers are to be changed.

The RD and WR lines determine whether the memory will be read or written, respectively. A read takes place if `RD = 1`, and a write takes place if `WR = 1`. Both the RD and WR fields cannot be set to 1 at the same time, but both fields can be 0 if neither a read nor a write operation is to take place. For both RD and WR, the address for the memory is taken directly from the A bus. The data input to the memory is taken from the B bus, and the data output from the memory is placed on the C bus. The RD line controls the 64-to-32 C Bus MUX, which determines whether the C bus is loaded from the memory (`RD = 1`) or from the ALU (`RD = 0`).

The ALU field determines which of the ALU operations is performed according to the settings shown in Figure 6-4. All 16 possible ALU field bit patterns correspond to valid ALU operations. This means that there is no way to “turn the ALU off” when it is not needed, such as during a read or write to memory. For this situation, an ALU operation should be selected that has no unwanted side effects. For example, `ANDCC` changes the condition codes and would not be appropriate, whereas the `AND` operation does not affect the condition codes, and would therefore be appropriate.

The COND (conditional jump) field instructs the **microcontroller** to take the next microword from either the next control store location, or from the location in the JUMP ADDR field of the MIR, or from the opcode bits of the instruction in `%ir`. The COND field is interpreted according to the table shown in Figure 6-12. If the COND field is 000, then no jump is taken, and the Next input to

C_2	C_1	C_0	Operation
0	0	0	Use NEXT ADDR
0	0	1	Use JUMP ADDR if $n = 1$
0	1	0	Use JUMP ADDR if $z = 1$
0	1	1	Use JUMP ADDR if $v = 1$
1	0	0	Use JUMP ADDR if $c = 1$
1	0	1	Use JUMP ADDR if $IR[13] = 1$
1	1	0	Use JUMP ADDR
1	1	1	DECODE

Figure 6-12 Settings for the COND field of the microword.

the CS Address MUX is used. The Next input to the CS Address MUX is com-

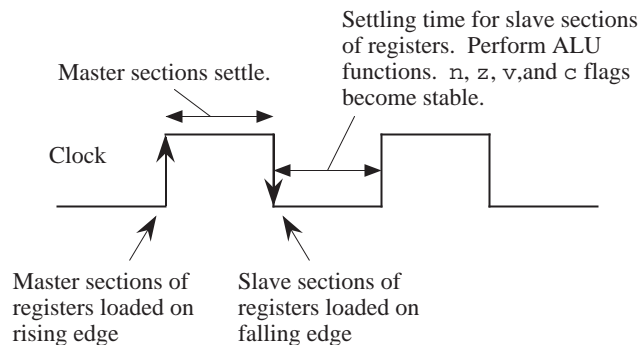


Figure 6-14 Timing relationships for the registers.

ing edge of the clock. On the rising edge of the clock, the new values of the registers are written into the master sections. The registers settle while the clock is high, and the process then repeats.

6.2.4 DEVELOPING THE MICROPROGRAM

In a microprogrammed architecture, instructions are interpreted by the microprogram in the control store. The microprogram is often referred to as **firmware** because it bridges the gap between the hardware and the software. The microarchitecture shown in Figure 6-10 needs firmware in order to execute ARC instructions, and one possible coding is described in this section.

A portion of a microprogram that implements the fetch-execute cycle for the ARC is shown in Figure 6-15. In the control store, each microstatement is stored in coded form (1's and 0's) in a single microword. For simplicity, the **micro-assembly language** shown in Figure 6-15 is loosely defined here, and we will leave out labels, pseudo-ops, *etc.*, that we would normally associate with a full-featured assembly language. Translation to the 41-bit format used in the microstore is not difficult to perform by hand for a small microprogram, and is frequently performed manually in practice (as we will do here) rather than creating a suite of software tools for such a small program.

Although our micro-assembly language is indeed an assembly language, it is not the same kind of assembly language as the ARC that we studied in Chapter 4. The ARC assembly language is visible to the user, and is used for coding general purpose programs. Our micro-assembly language is used for coding firmware, and is not visible to the user. The sole purpose of the firmware is to interpret a user-visible instruction set. A change to the instruction set involves changes to

Address	Operation Statements	Comment
0:	R[ir] ← AND(R[pc],R[pc]); READ;	/ Read an ARC instruction from main memory
1:	DECODE;	/ 256-way jump according to opcode
	/ sethi	
1152:	R[rd] ← LSHIFT10(ir); GOTO 2047;	/ Copy imm22 field to target register
	/ call	
1280:	R[15] ← AND(R[pc],R[pc]);	/ Save %pc in %r15
1281:	R[temp0] ← ADD(R[ir],R[ir]);	/ Shift disp30 field left
1282:	R[temp0] ← ADD(R[temp0],R[temp0]);	/ Shift again
1283:	R[pc] ← ADD(R[pc],R[temp0]);	/ Jump to subroutine
	GOTO 0;	
	/ addcc	
1600:	IF IR[13] THEN GOTO 1602;	/ Is second source operand immediate?
1601:	R[rd] ← ADDCC(R[rs1],R[rs2]);	/ Perform ADDCC on register sources
	GOTO 2047;	
1602:	R[temp0] ← SEXT13(R[ir]);	/ Get sign extended simm13 field
1603:	R[rd] ← ADDCC(R[rs1],R[temp0]);	/ Perform ADDCC on register/simm13 sources
	GOTO 2047;	
	/ andcc	
1604:	IF IR[13] THEN GOTO 1606;	/ Is second source operand immediate?
1605:	R[rd] ← ANDCC(R[rs1],R[rs2]);	/ Perform ANDCC on register sources
	GOTO 2047;	
1606:	R[temp0] ← SIMM13(R[ir]);	/ Get simm13 field
1607:	R[rd] ← ANDCC(R[rs1],R[temp0]);	/ Perform ANDCC on register/simm13 sources
	GOTO 2047;	
	/ orcc	
1608:	IF IR[13] THEN GOTO 1610;	/ Is second source operand immediate?
1609:	R[rd] ← ORCC(R[rs1],R[rs2]);	/ Perform ORCC on register sources
	GOTO 2047;	
1610:	R[temp0] ← SIMM13(R[ir]);	/ Get simm13 field
1611:	R[rd] ← ORCC(R[rs1],R[temp0]);	/ Perform ORCC on register/simm13 sources
	GOTO 2047;	
	/ orncc	
1624:	IF IR[13] THEN GOTO 1626;	/ Is second source operand immediate?
1625:	R[rd] ← NORCC(R[rs1],R[rs2]);	/ Perform ORNCC on register sources
	GOTO 2047;	
1626:	R[temp0] ← SIMM13(R[ir]);	/ Get simm13 field
1627:	R[rd] ← NORCC(R[rs1],R[temp0]);	/ Perform NORCC on register/simm13 sources
	GOTO 2047;	
	/ srl	
1688:	IF IR[13] THEN GOTO 1690;	/ Is second source operand immediate?
1689:	R[rd] ← SRL(R[rs1],R[rs2]);	/ Perform SRL on register sources
	GOTO 2047;	
1690:	R[temp0] ← SIMM13(R[ir]);	/ Get simm13 field
1691:	R[rd] ← SRL(R[rs1],R[temp0]);	/ Perform SRL on register/simm13 sources
	GOTO 2047;	
	/ jmp1	
1760:	IF IR[13] THEN GOTO 1762;	/ Is second source operand immediate?
1761:	R[pc] ← ADD(R[rs1],R[rs2]);	/ Perform ADD on register sources
	GOTO 0;	

Figure 6-15 Partial microprogram for the ARC. Microwords are shown in logical sequence (not numerical sequence.)

the firmware, whereas a change in user-level software has no influence on the firmware.

Each statement in the microprogram shown in Figure 6-15 is preceded by a decimal number that indicates the address of the corresponding microword in the 2048-word control store. The address is followed by a colon. The operation statements follow the address, and are terminated by semicolons. An optional

```

1762: R[temp0] ← SEXT13(R[ir]);           / Get sign extended simm13 field
1763: R[pc] ← ADD(R[rs1],R[temp0]);        / Perform ADD on register/simm13 sources
      GOTO 0;
      / ld
1792: R[temp0] ← ADD(R[rs1],R[rs2]);       / Compute source address
      IF IR[13] THEN GOTO 1794;
1793: R[rd] ← AND(R[temp0],R[temp0]);     / Place source address on A bus
      READ; GOTO 2047;
1794: R[temp0] ← SEXT13(R[ir]);           / Get simm13 field for source address
1795: R[temp0] ← ADD(R[rs1],R[temp0]);    / Compute source address
      GOTO 1793;
      / st
1808: R[temp0] ← ADD(R[rs1],R[rs2]);       / Compute destination address
      IF IR[13] THEN GOTO 1810;
1809: R[ir] ← RSHIFT5(R[ir]); GOTO 40;    / Move rd field into position of rs2 field
      40: R[ir] ← RSHIFT5(R[ir]);          / by shifting to the right by 25 bits.
      41: R[ir] ← RSHIFT5(R[ir]);
      42: R[ir] ← RSHIFT5(R[ir]);
      43: R[ir] ← RSHIFT5(R[ir]);
      44: R[0] ← AND(R[temp0], R[rs2]);     / Place destination address on A bus and
      WRITE; GOTO 2047;                  / place operand on B bus
1810: R[temp0] ← SEXT13(R[ir]);           / Get simm13 field for destination address
1811: R[temp0] ← ADD(R[rs1],R[temp0]);    / Compute destination address
      GOTO 1809;
      / Branch instructions: ba, be, bcs, bvs, bneg
1088: GOTO 2;                             / Decoding tree for branches
      2: R[temp0] ← LSHIFT10(R[ir]);       / Sign extend the 22 LSB's of %temp0
      3: R[temp0] ← RSHIFT5(R[temp0]);    / by shifting left 10 bits, then right 10
      4: R[temp0] ← RSHIFT5(R[temp0]);    / bits. RSHIFT5 does sign extension.
      5: R[ir] ← RSHIFT5(R[ir]);          / Move COND field to IR[13] by
      6: R[ir] ← RSHIFT5(R[ir]);          / applying RSHIFT5 three times. (The
      7: R[ir] ← RSHIFT5(R[ir]);          / sign extension is inconsequential.)
      8: IF IR[13] THEN GOTO 12;          / Is it ba?
      R[ir] ← ADD(R[ir],R[ir]);
      9: IF IR[13] THEN GOTO 13;          / Is it not be?
      R[ir] ← ADD(R[ir],R[ir]);
      10: IF Z THEN GOTO 12;              / Execute be
      R[ir] ← ADD(R[ir],R[ir]);
      11: GOTO 2047;                     / Branch for be not taken
      12: R[pc] ← ADD(R[pc],R[temp0]);    / Branch is taken
      GOTO 0;
      13: IF IR[13] THEN GOTO 16;        / Is it bcs?
      R[ir] ← ADD(R[ir],R[ir]);
      14: IF C THEN GOTO 12;              / Execute bcs
      15: GOTO 2047;                     / Branch for bcs not taken
      16: IF IR[13] THEN GOTO 19;        / Is it bvs?
      17: IF N THEN GOTO 12;              / Execute bneg
      18: GOTO 2047;                     / Branch for bneg not taken
      19: IF V THEN GOTO 12;              / Execute bvs
      20: GOTO 2047;                     / Branch for bvs not taken
2047: R[pc] ← INCP(R[pc]); GOTO 0;       / Increment %pc and start over

```

Figure 6-15 (cont').

comment follows the operation field and begins with a slash '/.' The comment terminates at the end of the line. More than one operation is allowed per line, as long as all of the operations can be performed in a single instruction cycle. The ALU operations come from Figure 6-4, and there are a few others as we will see. Note that the 65 statements are shown in logical sequence, rather than in numerical sequence.

Before the microprogram begins execution, the PC is set up with the starting address of a program that has been loaded into the main memory. This may hap-

pen as the result of an initialization sequence when the computer is powered on, or by the operating system during the normal course of operation.

The first task in the execution of a user-level program is to bring the instruction pointed to by the PC from the main memory into the IR. Recall from Figure 6-10 that the address lines to main memory are taken from the A bus. In line 0, the PC is loaded onto the A bus, and a Read operation is initiated to memory. The notation “R[x]” means “register x,” in which x is replaced with one of the registers in the datapath, and so “R[1]” means “register %r1,” “R[ir]” means “register %ir,” and “R[rs1]” means the register that appears in the 5-bit rs1 field of an instruction (refer to Figure 6-2.)

The expression “AND(R[pc],R[pc])” simply performs a logical AND of %pc with itself in a literal interpretation. This operation is not very useful in a logical sense, but what we are interested in are the side effects. In order to place %pc onto the A bus, we have to choose an ALU operation that uses the A bus but does not affect the condition codes. There is a host of alternative choices that can be used, and the AND approach is arbitrarily chosen here. Note that the result of the AND operation is discarded because the C bus MUX in Figure 6-10 only allows the data output from main memory onto the C bus during a read operation.

A read operation normally takes more time to complete than the time required for one microinstruction to execute. The access time of main memory can vary depending on the memory organization, as we will see in Chapter 7. In order to account for variations in the access times of memory, the control store address incrementer (CSAI) does not increment the address until an acknowledge (ACK) signal is sent which indicates the memory has completed its operation.

Flow of control within the microprogram defaults to the next higher numbered statement unless a GOTO operation or a DECODE operation is encountered, and so microword 1 (line 1) is read into the MIR on the next cycle. Notice that some of the microcode statements in Figure 6-15 take up more than one line on the page, but are part of a single microinstruction. See, for example, lines 1283 and 1601.

Now that the instruction is in the IR as a result of the read operation in line 0, the next step is to decode the opcode fields. This is performed by taking a 256-way branch into the microcode as indicated by the DECODE keyword in line 1 of the microprogram. The 11-bit pattern for the branch is constructed by

appending a 1 to the left of bits 30 and 31 of the IR, followed by bits 19-24 of the IR, followed by the pattern 00. After the opcode fields are decoded, execution of the microcode continues according to which of the 15 ARC instructions is being interpreted.

As an example of how the decode operation works, consider the instruction. According to the Arithmetic instruction format in Figure 6-2, the op field is 10 and the $op3$ field is 010000. If we append a 1 to the left of the pattern, followed by the $op3$ bit pattern, followed by 00, the $DECODE$ address is $11001000000 = (1600)_{10}$. This means that the microinstructions that interpret the $addcc$ instruction begin at control store location 1600.

A number of $DECODE$ addresses should never arise in practice. There is no Arithmetic instruction that corresponds to the invalid field 111111, but if this situation does arise, possibly due to an errant program, then a microstore routine should be placed at the corresponding $DECODE$ address $1101111100 = (1788)_{10}$ in order to deal with the illegal instruction. These locations are left blank in the microprogram shown in Figure 6-15.

Instructions in the SETHI/Branch and Call formats do not have $op3$ fields. The SETHI/Branch formats have op and $op2$ fields, and the Call format has only the op field. In order to maintain a simple decoding mechanism, we can create duplicate entries in the control store. Consider the SETHI format. If we follow the rule for constructing the $DECODE$ address, then the $DECODE$ address will have a 1 in the leftmost position, followed by 00 for the op field, followed by 100 which identifies SETHI in bit positions 19 \div 21, followed by the bits in positions 22 \div 24 of the IR, followed by 00, resulting in the bit pattern $100100xxx00$ where xxx can take on any value, depending on the $op2$ field. There are eight possible bit patterns for the xxx bits, and so we need to have duplicate SETHI codes at locations 1000000 , 1001000100 , 1001001000 , 1001001100 , 1001001000 , 1001001000 , 1001001000 , and 1001001100 . $DECODE$ addresses for the Branch and CALL formats are constructed in duplicate locations in a similar manner. Only the lowest addressed version of each set of duplicate codes is shown in Figure 6-15.

Although this method of decoding is fast and simple, a large amount of control store memory is wasted. An alternative approach that wastes much less space is to modify the decoder for the control store so that all possible branch patterns for SETHI point to the same location, and the same for the Branch and Call format instructions. For our microarchitecture, we will stay with the simpler approach

and pay the price of having a large control store.

Consider now how the `ld` instruction is interpreted. The microprogram begins at location 0, and at this point does not know that `ld` is the instruction that the PC points to in main memory. Line 0 of the microprogram begins the Read operation as indicated by the `READ` keyword, which brings an instruction into the IR from the main memory address pointed to by the PC. For this case, let us assume that the IR now contains the 32-bit pattern:

```
11 00010 000000 00101 1 0000001010000
op  rd    op3   rs1  i   simm13
```

which is a translation of the ARC assembly code: `ld %r5 + 80, %r2`. Line 1 then performs a branch to control store address $(11100000000)_2 = (1792)_{10}$.

At line 1792, execution of the `ld` instruction begins. In line 1792, the immediate bit `i` is tested. For this example, `i` = 1, and so control is transferred to microword 1794. If instead we had `i` = 0, then control would pass to the next higher numbered microword, which is 1793 for this case. Line 1792 adds the registers in the `rs1` and `rs2` fields of the instruction, in anticipation of a non-immediate form of `ld`, but this only makes sense if `i` = 0, which it is not for this example. The result that is stored in `%temp0` is thus discarded when control is transferred to microword 1794, but this introduces no time penalty and does not produce any unwanted side effects (`ADD` does not change the condition codes).

In microword 1794, the `simm13` field is extracted (using sign extension, as indicated by the `SEXT13` operation), which is added with the register in the `rs1` field in microword 1795. Control is then passed to microword 1793 which is where the `READ` operation takes place. Control passes to line 2047 where the PC is incremented in anticipation of reading the next instruction from main memory. Since instructions are four bytes long and must be aligned on word boundaries in memory, the PC is incremented by four. Control then returns to line 0 where the process repeats. A total of seven microinstructions are thus executed in

interpreting the `ld` instruction. These microinstructions are repeated below:

```

0: R[ir] ← AND(R[pc],R[pc]); READ;      / Read an ARC instruction from main memory.
1: DECODE;                               / 256-way jump according to opcode
1792: R[temp0] ← ADD(R[rs1],R[rs2]);      / Compute source address
      IF IR[13] THEN GOTO 1794;
1794: R[temp0] ← SEXT13(R[ir]);           / Get simm13 field for source address
1795: R[temp0] ← ADD(R[rs1],R[temp0]);    / Compute source address
      GOTO 1793;
1793: R[rd] ← AND(R[temp0],R[temp0]);     / Place source address on A bus
      READ; GOTO 2047;
2047: R[pc] ← INCPC(R[pc]); GOTO 0;      / Increment %pc and start over

```

The remaining instructions, except for branches, are interpreted similar to the way `ld` is interpreted. Additional decoding is needed for the branch instructions because the type of branch is determined by the `COND` field of the branch format (bits 25 – 28), which is not used during a `DECODE` operation. The approach used here is to shift the `COND` bits into `IR[13]` one bit at a time, and then jump to different locations in the microcode depending on the `COND` bit pattern.

For branch instructions, the `DECODE` operation on line 2 of the microprogram transfers control to location 1088. We need more space for the branch instructions than the four-word per instruction allocation, so line 1088 transfers control to line 2 which is the starting address of a large section of available control store memory.

Lines 2 – 4 extract the 22-bit displacement for the branch by zeroing the high order 10 bits and storing the result in `%temp0`. This is accomplished by shifting `%ir` to the left by 10 bits and storing it in `%temp0`, and then shifting the result back to the right by 10 bits. (Notice that sign extension should be performed on the displacement, which may be negative. We will leave it as it is to simplify the discussion.) Lines 5 – 7 shift `%ir` to the right by 15 bits so that the most significant `COND` bit (`IR[28]`) lines up in position `IR[13]`, which allows the `Jump on IR[13]=1` operation to test each bit. Alternatively, we could shift the `COND` field to `IR[31]` one bit at a time, and use the `Jump on n` condition to test each bit.

Line 8 starts the branch decoding process, which is summarized in Figure 6-16. If `IR[28]`, which is now in `IR[13]`, is set to 1, then the instruction is `ba`, which is executed in line 12. Notice that control returns to line 0, rather than to line 2047, so that the PC does not get changed twice for the same instruction.

If `IR[28]` is zero, then `%ir` is shifted to the left by one bit by adding it to itself,

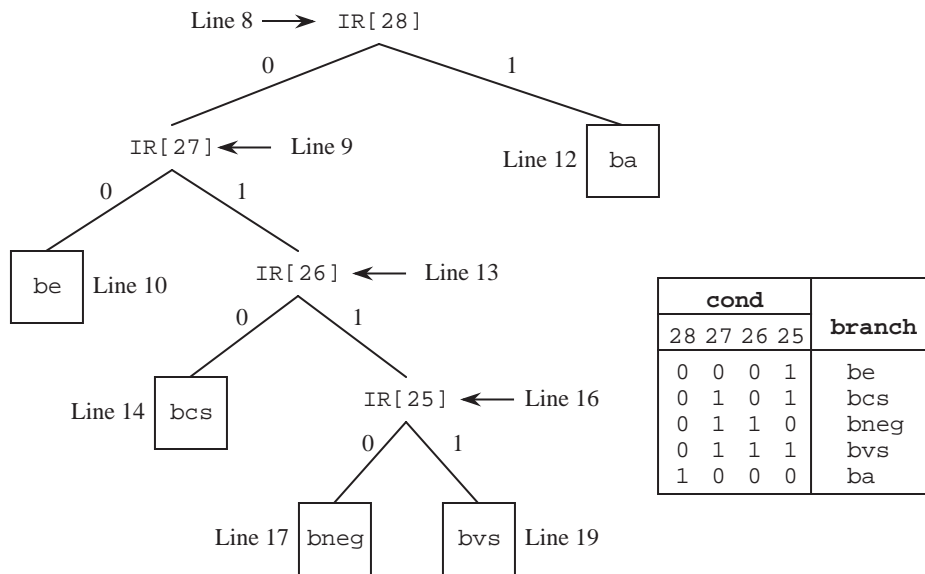


Figure 6-16 Decoding tree for branch instructions, showing corresponding microprogram lines.

so that $IR[27]$ lines up in position $IR[13]$. Bit $IR[27]$ is tested in line 9. If $IR[27]$ is zero, then the `be` instruction is executed in line 10, otherwise $\%ir$ is shifted to the left and $IR[26]$ is then tested in line 13. The remaining branch instructions are interpreted in a similar manner.

Microassembly Language Translation

A microassembly language microprogram must be translated into binary object code before it is stored in the control store, just as an assembly language program must be translated into a binary object form before it is stored in main memory. Each line in the ARC microprogram corresponds to exactly one word in the control store, and there are no unnumbered forward references in the microprogram, so we can assemble the ARC microprogram one line at a time in a single pass. Consider assembling line 0 of the microprogram shown in Figure 6-15:

```
0: R[ir] ← AND(R[pc],R[pc]); READ;
```

We can fill in the fields of the 41-bit microword as shown below:

A					B					C					JUMP ADDR				
M					M					M									
U					U					U R W									
X					X					X D R									
1	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0

The PC is enabled onto both the A and B busses for the AND operation, which transfers a word through the ALU without changing it. The A and B fields have the bit pattern for the PC ($32_{10} = 100000_2$). The AMUX and BMUX fields both contain 0's, since the inputs to these MUXes are taken from the MIR. The target of the Read operation is the IR, which has a corresponding bit pattern of ($37_{10} = 100101_2$) for the C field. The CMUX field contains a 0 because the input to the CMUX is taken from the MIR. A read operation to memory takes place, and so the RD field contains a 1 and the WR field contains a 0. The ALU field contains 0101, which corresponds to the AND operation. Note that the condition codes are not affected, which would happen if ANDCC is used instead. The COND field contains 000 since control passes to the next microword, and so the bit pattern in the JUMP ADDR field does not matter. Zeros are arbitrarily placed in the JUMP ADDR field.

The second microword implements the 256-way branch. For this case, all that matters is that the bit pattern 111 appears in the COND field for the DECODE operation, and that no registers, memory, or condition codes are disturbed. The corresponding bit pattern is then:

A					B					C					JUMP ADDR				
M					M					M									
U					U					U R W									
X					X					X D R									
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

A number of different bit patterns would also work for line 1. For example, any bit patterns can appear in the A, B, or JUMP ADDR fields when a DECODE operation takes place. The use of the zero bit patterns is an arbitrary choice. The ALU field is 0101 which is for AND, which does not affect the condition codes. Any other ALU operation that does not affect the condition codes can also be used.

The remainder of the microprogram is translated in a similar manner. The trans-

Microstore Address	A		B		C					
	M		M		M					
	U		U		U	RW				
	A	X	B	X	C	XDR	ALU	COND	JUMP ADDR	
0	1000000	01000000	01000000	01001010	01001010	000	000000000000			
1	0000000	00000000	00000000	00000000	00001011	111	000000000000			
1152	1001010	00000000	00000000	00000000	10010101	110	111111111111			
1280	1000000	01000000	00011111	00001010	00001010	000	000000000000			
1281	1001010	01001010	01000001	00010000	00010000	000	000000000000			
1282	1000010	01000001	01000001	00010000	00010000	000	000000000000			
1283	1000000	01000001	01000000	00010000	00010000	110	000000000000			
1600	0000000	00000000	00000000	00000000	00001011	101	110010000010			
1601	0000000	10000000	10000000	10000000	10000011	110	111111111111			
1602	1001010	00000000	01000001	00001100	00001100	000	000000000000			
1603	0000000	11000001	00000000	10000011	00000111	110	111111111111			
1604	0000000	00000000	00000000	00000000	00001011	101	110010000110			
1605	0000000	10000000	10000000	10000000	10000000	110	111111111111			
1606	1001010	00000000	01000001	00001011	00001011	000	000000000000			
1607	0000000	11000001	00000000	00000000	10000000	110	111111111111			
1608	0000000	00000000	00000000	00000000	00001011	101	110010001010			
1609	0000000	10000000	10000000	10000000	10000001	110	111111111111			
1610	1001010	00000000	01000001	00001011	00001011	000	000000000000			
1611	0000000	11000001	00000000	10000001	00000011	110	111111111111			
1624	0000000	00000000	00000000	00000000	00001011	101	1100101101010			
1625	0000000	10000000	10000000	10000000	10000010	110	111111111111			
1626	1001010	00000000	01000001	00001011	00001011	000	000000000000			
1627	0000000	11000001	00000000	10000001	00000101	110	111111111111			
1688	0000000	00000000	00000000	00000000	00000101	000	1101001101010			
1689	0000000	10000000	10000000	10000000	10000100	110	111111111111			
1690	1001010	00000000	01000001	00001011	00001011	000	000000000000			
1691	0000000	11000001	00000000	10000001	00001000	110	111111111111			
1760	0000000	00000000	00000000	00000000	00000101	101	110111000010			
1761	0000000	10000000	10000000	10000000	00010000	110	000000000000			
1762	1001010	00000000	01000001	00001100	00001100	000	000000000000			
1763	0000000	11000001	01000000	00001000	00001000	110	000000000000			
1792	0000000	10000000	10000001	00001000	00001000	101	111000000001			

Figure 6-17 Assembled microprogram for the ARC instruction subset.

lated microprogram is shown in Figure 6-17, except for gaps where duplicate branch code would appear, or where “illegal instruction” code would appear.

EXAMPLE

Consider adding an instruction called `subcc` to the microcoded implementation of the ARC instruction set, which subtracts its second source operand from the first, using two's complement arithmetic. The new instruction uses the Arithmetic format and an `op3` field of 001100.

	A		B		C		ALU		COND	JUMP	ADDR
	A	X	B	X	C	X	D	R			
1793	1	0	0	0	0	1	0	0	1	0	1
1794	1	0	0	1	0	1	0	0	0	0	0
1795	0	0	0	0	0	1	0	0	0	1	1
1808	0	0	0	0	0	1	0	0	0	1	0
1809	1	0	0	1	0	1	0	0	1	1	0
40	1	0	0	1	0	1	0	0	1	1	1
41	1	0	0	1	0	1	0	0	1	1	1
42	1	0	0	1	0	1	0	0	1	1	1
43	1	0	0	1	0	1	0	0	1	1	1
44	1	0	0	0	0	1	0	0	0	0	0
1810	1	0	0	1	0	1	0	0	1	1	0
1811	0	0	0	0	0	1	0	0	0	1	1
1088	0	0	0	0	0	0	0	0	0	1	0
2	1	0	0	1	0	1	0	0	1	0	1
3	1	0	0	0	0	1	0	0	1	1	1
4	1	0	0	0	0	1	0	0	1	1	1
5	1	0	0	1	0	1	0	0	1	1	1
6	1	0	0	1	0	1	0	0	1	1	1
7	1	0	0	1	0	1	0	0	1	1	1
8	1	0	0	1	0	1	0	0	1	0	0
9	1	0	0	1	0	1	0	0	1	0	1
10	1	0	0	1	0	1	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	1	1
12	1	0	0	0	0	1	0	0	1	0	0
13	1	0	0	1	0	1	0	0	1	0	0
14	0	0	0	0	0	0	0	0	0	1	0
15	0	0	0	0	0	0	0	0	0	1	1
16	0	0	0	0	0	0	0	0	0	1	0
17	0	0	0	0	0	0	0	0	0	1	0
18	0	0	0	0	0	0	0	0	0	1	1
19	0	0	0	0	0	0	0	0	0	1	0
20	0	0	0	0	0	0	0	0	0	1	1
2047	1	0	0	0	0	1	0	0	1	1	0

Figure 6-17 (Continued.)

We need to modify the microprogram to add this new instruction. We start by computing the starting location of `subcc` in the control store, by appending a '1' to the left of the `op` field, which is 10, followed by the `op3` field which is 001100, followed by 00. This results in the bit pattern 11000110000 which corresponds to control store location $(1584)_{10}$. We can then create microassembly code that is similar to the `addcc` microassembly code at location 1600, except that the two's complement negative of the subtrahend (the second source operand) is formed before performing the addition. The subtrahend is complemented by making use of the `NOR` operation, and 1 is added to it by using the `INC` operation. The subtraction is then completed by using the code for `addcc`. A

microassembly coding for `subcc` is shown below:

```

1584: R[temp0] ← SEXT13(R[ir]);           / Extract rs2 operand
      IF IR[13] THEN GOTO 1586;           / Is second source immediate?
1585: R[temp0] ← R[rs2];                 / Extract sign extended immediate operand
1586: R[temp0] ← NOR(R[temp0], R[0]);      / Form one's complement of subtrahend
1587: R[temp0] ← INC(R[temp0]); GOTO 1603; / Form two's complement of subtrahend

```

The corresponding microcode for one possible translation is then:

	A		B		C		XDR		ALU	COND	JUMP ADDR	
	A	X	B	X	C	X	D	R				
1584	1	0	0	1	0	1	0	0	0	0	0	0
1585	0	0	0	0	0	0	0	0	1	0	0	0
1586	1	0	0	0	0	1	0	0	0	1	1	0
1587	1	0	0	0	0	1	0	0	0	1	1	0

6.2.5 TRAPS AND INTERRUPTS

A **trap** is an automatic procedure call initiated by the hardware after an exceptional condition caused by an executing program, such as an illegal instruction, overflow, underflow, dividing by zero, *etc.* When a trap occurs, control is transferred to a “trap handler” which is a routine that is part of the operating system. The handler might do something like print a message and terminate the offending program.

One way to handle traps is to modify the microcode, possibly to check the status bits. For instance, we can check the v bit to see if an overflow has occurred. The microcode can then load an address into the PC (if a trap occurs) for the starting location of the trap handler.

Normally, there is a fixed section of memory for trap handler starting addresses where only a single word is allocated for each handler. This section of memory forms a **branch table** that transfers control to the handlers, as illustrated in Figure 6-18. The reason for using a branch table is that the absolute addresses for each type of trap can be embedded in the microcode this way, while the targets of the jumps can be changed at the user level to handle traps differently.

Address	Contents	Trap Handler
	⋮	
60	JUMP TO 2000	Illegal instruction
64	JUMP TO 3000	Overflow
68	JUMP TO 3600	Underflow
72	JUMP TO 5224	Zerodivide
76	JUMP TO 4180	Disk
80	JUMP TO 5364	Printer
84	JUMP TO 5908	TTY
88	JUMP TO 6048	Timer
	⋮	

Figure 6-18 A branch table for trap handlers and interrupt service routines.

A historically common trap is for floating point instructions, which may be **emulated** by the operating system if they are not implemented directly in hardware. Floating point instructions have their own opcodes, but if they are not implemented by the hardware (that is, the microcode does not know about them) then they will generate an illegal instruction trap when an attempt is made to execute them. When an illegal instruction occurs, control is passed to the illegal instruction handler which checks to see if the trap is caused by a floating point instruction, and then passes control to a floating point emulation routine as appropriate for the cause of the trap. Although floating point units are normally integrated into CPU chips these days, this method is still used when extending the instruction set for other instructions, such as graphics extensions to the ISA.

Interrupts are similar to traps, but are initiated after a hardware **exception** such as a user hitting a key on a keyboard, an incoming telephone call for a modem, a power fluctuation, an unsafe operating temperature, *etc.* Traps are *synchronous* with a running program, whereas interrupts are *asynchronous*. Thus, a trap will always happen at the same place in the same program running with the same data set, whereas the timing of interrupts is largely unpredictable.

When a key is pressed on an interrupt based keyboard, the keyboard asserts an interrupt line on the bus, and the CPU then asserts an acknowledge line as soon as it is ready (this is where **bus arbitration** comes in, which is covered in Chapter 8, if more than one device wants to interrupt at the same time). The keyboard then places an **interrupt vector** onto the data bus which identifies itself to the

CPU. The CPU then pushes the program counter and processor status register (where the flags are stored) onto the stack. The interrupt vector is used to index into the branch table, which lists the starting addresses of the interrupt service routines.

When a trap handler or an interrupt service routine begins execution, it saves the registers that it plans to modify on the stack, performs its task, restores the registers, and then returns from the interrupt. The process of returning from a trap is different from returning from a subroutine, since the process of entering a trap is different from a subroutine call (because the `%psr` register is also saved and restored). For the ARC, the `rti` instruction (see Chapter 8) is used for returning from a trap or interrupt. Interrupts can interrupt other interrupts, and so the first thing that an interrupt service routine might do is to raise its priority (using a special **supervisor mode** instruction) so that no interrupts of lower priority are accepted.

6.2.6 NANOPROGRAMMING

If the microstore is wide, and has lots of the same words, then we can save microstore memory by placing one copy of each unique microword in a **nanostore**, and then use the microstore to index into the nanostore. For instance, in the microprogram shown in Figure 6-15, lines 1281 and 1282 are the same. Lines 3, 4, and 40-44 are the same, and there are a number of other microinstructions that recur, especially for the duplicated branch microcode and the duplicated illegal instruction microcode.

Figure 6-19a illustrates the space requirement for the original microstore ROM. There are $n=2048$ words that are each 41 bits wide, giving an area complexity of $2048 \times 41 = 83,968$ bits. Suppose now that there are 100 unique microwords in the ROM (the microprogram in Figure 6-15 is only partially complete so we cannot measure the number of unique microwords directly). Figure 6-19b illustrates a configuration that uses a nanostore, in which an area savings can be realized if there are a number of bit patterns that recur in the original microcode sequence. The unique microwords (100 for this case) form a nanoprogram, which is stored in a ROM that is 100 words deep by 41 bits wide.

The microprogram now indexes into the nanostore. The microprogram has the same number of microwords regardless of whether or not a nanostore is used, but when a nanostore is used, *pointers* into the nanostore are stored in the microstore rather than the wider 41-bit words. For this case, the microstore is now 2048

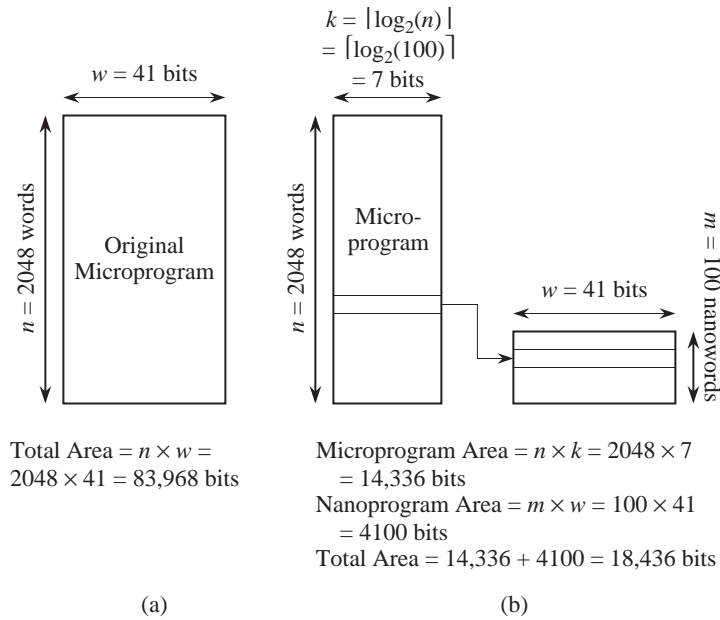


Figure 6-19 (a) Microprogramming vs (b) nanoprogramming.

words deep by $\lceil \log_2(100) \rceil = 7$ bits wide. The area complexity using a nanostore is then $100 \times 41 + 2048 \times 7 = 18,436$ bits, which is a considerable savings in area over the original microcoded approach.

For small m and large n , where m is the length of the nanoprogram, we can realize a large savings in memory. This frees up area that can be applied in some other way, possibly to improve performance. However, instead of accessing only the microstore, we must now access the microstore first, followed by an access to the nanostore. The machine will thus run more slowly, but will fit into a smaller area.

6.3 Hardwired Control

An alternative approach to a microprogrammed control unit is to use a **hard-wired** approach, in which a direct implementation is created using flip-flops and logic gates, instead of using a control store and a microword selection mechanism. States in a finite state machine replace steps in the microprogram.

In order to manage the complexity of design for a hardwired approach, a **hardware description language** (HDL) is frequently used to represent the control

structure. One example of an HDL is **VHDL**, which is an acronym for **VHSIC Hardware Description Language** (in which VHSIC is yet another acronym for **Very High Speed Integrated Circuit**). VHDL is used for describing an architecture at a very high level, and can be compiled into hardware designs through a process known as **silicon compilation**. For the hardwired control unit we will design here, a lower level HDL that is sometimes referred to as a **register transfer language** (RTL) is more appropriate.

We will define a simple HDL/RTL in this section that loosely resembles Hill & Peterson's **A Hardware Programming Language** (AHPL) (Hill and Peterson, 1987). The general idea is to express a control sequence as a series of numbered statements, which can then be directly translated into a hardware design. Each statement consists of a data portion and a transfer portion, as shown below:

```
5: A ← ADD(B,C);           ! Data portion
    GOTO {10 CONDITIONED ON IR[12]}. ! Control portion
```

The statement is labelled “5,” which means that it is preceded by statement 4 and is succeeded by statement 6, unless an out-of-sequence transfer of control takes place. The left arrow (\leftarrow) indicates a data transfer, to register A for this case. The “ADD(B,C)” construct indicates that registers B and C are sent to a combinational logic unit (CLU) that performs the addition. Comments begin with an exclamation mark (!) and terminate at the end of the line. The GOTO construct indicates a transfer of control. For this case, control is transferred to statement 10 if bit 12 of register IR is true, otherwise control is transferred to the next higher numbered statement (6 for this case).

Figure 6-20 shows an HDL description of a modulo 4 counter. The counter produces the output sequence: 00, 01, 10, 11 and then repeats as long as the input line x is 0. If the input line is set to 1, then the counter returns to state 0 at the end of the next clock cycle. The comma is the catenation operator, and so the statement “ $Z \leftarrow 0, 0;$ ” assigns the two-bit pattern 00 to the two-bit output Z.

The HDL sequence is composed of three sections: the *preamble*, the *numbered statements*, and the *epilogue*. The preamble names the module with the “MODULE” keyword and declares the inputs with the “INPUTS” keyword, the outputs with the “OUTPUTS” keyword, and the arity (number of signals) of both, as well as any additional storage with the “MEMORY” keyword (none for this example). The numbered statements follow the preamble. The epilogue closes the sequence with the key phrase “END SEQUENCE.” The key phrase “END

Preamble	{	MODULE: MOD_4_COUNTER.
		INPUTS: x.
		OUTPUTS: Z[2].
		MEMORY:
	}	
Statements	{	0: Z ← 0,0;
		GOTO {0 CONDITIONED ON x,
		1 CONDITIONED ON \bar{x} }.
		1: Z ← 0,1;
		GOTO {0 CONDITIONED ON x,
		2 CONDITIONED ON \bar{x} }.
		2: Z ← 1,0;
		GOTO {0 CONDITIONED ON x,
		3 CONDITIONED ON \bar{x} }.
		3: Z ← 1,1;
		GOTO 0.
	}	
Epilogue	{	END SEQUENCE.
		END MOD_4_COUNTER.
	}	

Figure 6-20 HDL sequence for a resettable modulo 4 counter.

MOD_4_COUNTER” closes the description of the module. Anything that appears between “END SEQUENCE” and “END MOD_4_COUNTER” occurs *continuously*, independent of the statement number. There are no such statements for this case.

In translating an HDL description into a design, the process can be decomposed into separate parts for the control section and the data section. The control section deals with how transitions are made from one statement to another. The data section deals with producing outputs and changing the values of any memory elements.

We consider the control section first. There are four numbered statements, and so we will use four flip-flops, one for each statement, as illustrated in Figure 6-21. This is referred to as a **one-hot encoding** approach, because exactly one flip-flop holds a true value at any time. Although four states can be encoded using only two flip-flops, studies have shown that the one-hot encoding approach results in approximately the same circuit area when compared with a more densely encoded approach; but more importantly, the complexity of the transfers from one state to the next are generally simpler and can be implemented with shallow combinational logic circuits, which means that the clock rate can be faster for a one-hot encoding approach than for a densely encoded approach.

In designing the control section, we first draw the flip-flops, apply labels as

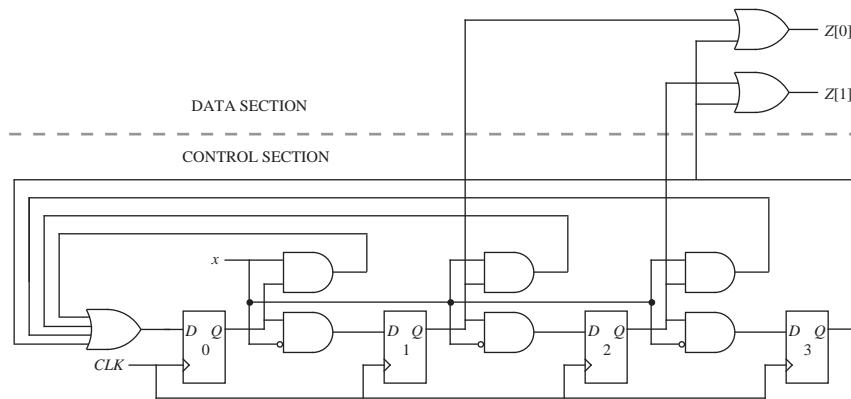


Figure 6-21 Logic design for a modulo 4 counter described in HDL.

appropriate, and connect the clock inputs. The next step is to simply scan the numbered statements in order and add logic as appropriate for the transitions. From statement 0, there are two possible transitions to statements 0 or 1, conditioned on x or its complement, respectively. The output of flip-flop 0 is thus connected to the inputs of flip-flops 0 and 1, through AND gates that take the value of the x input into account. Note that the AND gate into flip-flop 1 has a circle at one of its inputs, which is a simplified notation that means x is complemented by an inverter before entering the AND gate.

A similar arrangement of logic gates is applied for statements 1 and 2, and no logic is needed at the output of flip-flop 3 because statement 3 returns to statement 1 unconditionally. The control section is now complete and can execute correctly on its own. No outputs are produced, however, until the data section is implemented.

We now consider the design of the data section, which is trivial for this case. Both bits of the output Z change in every statement, and so there is no need to condition the generation of an output on the state. We only need to produce the correct output values for each of the statements. The least significant bit of Z is true in statements 1 and 3, and so the outputs of the corresponding control flip-flops are ORed to produce $Z[0]$. The most significant bit of Z is true in statements 2 and 3, and so the outputs of the corresponding control flip-flops are ORed to produce $Z[1]$. The entire circuit for the mod 4 counter is now complete, as shown in Figure 6-21.

We can now use our HDL in describing the control section of the ARC microarchitecture. There is no need to design the data section, since we have already defined its form in Figure 6-10. The data section is the same for both the microcoded and hardwired approaches. As for the microcoded approach, the operations that take place for a hardwired approach are:

- 1) Fetch the next instruction to be executed from memory.
- 2) Decode the opcode.
- 3) Read operand(s) from main memory, if any.
- 4) Execute the instruction and store results.
- 5) Go to Step 1.

The microcode of Figure 6-15 can serve as a guide for what needs to be done. The first step is to fetch the next user-level instruction from main memory. The following HDL line describes this operation:

```
0: ir ← AND(pc, pc); Read = 1.
```

The structure of this statement is very similar to the first line of the microprogram, which may not be surprising since the same operations must be carried out on the same datapath.

Now that the instruction has been fetched, the next operation is to decode the opcode. This is where the power of a hardwired approach comes into play. Since every instruction has an `op` field, we can decode that field first, and then decode the `op2`, `op3`, and `cond` fields as appropriate for the instruction.

The next line of the control sequence decodes the `op` field:

```
1: GOTO {2 CONDITIONED ON  $\overline{\text{IR}[31]} \times \overline{\text{IR}[30]}$ , ! Branch/Sethi format: op=00
        4 CONDITIONED ON  $\overline{\text{IR}[31]} \times \text{IR}[30]$ , ! Call format: op=01
        8 CONDITIONED ON  $\text{IR}[31] \times \overline{\text{IR}[30]}$ , ! Arithmetic format: op=10
       10 CONDITIONED ON  $\text{IR}[31] \times \text{IR}[30]$ }. ! Memory format: op=11
```

The product symbol “ \times ” indicates a logical AND operation. Control is thus

transferred to one of the four numbered statements: 2, 4, 8, or 10 depending on the bit pattern in the `op` field.

Figure 6-24 shows a complete HDL description of the control section. We may

```

MODULE: ARC_CONTROL_UNIT.
INPUTS:
OUTPUTS: C, N, V, Z. ! These are set by the ALU
MEMORY: R[16][32], pc[32], ir[32], temp0[32], temp1[32], temp2[32],
        temp3[32].

0: ir ← AND(pc, pc); Read ← 1;          ! Instruction fetch
   ! Decode op field
1: GOTO {2 CONDITIONED ON ir[31]×ir[30], ! Branch/sethi format: op=00
        4 CONDITIONED ON ir[31]×ir[30], ! Call format: op=01
        8 CONDITIONED ON ir[31]×ir[30], ! Arithmetic format: op=10
        10 CONDITIONED ON ir[31]×ir[30]}. ! Memory format: op=11
   ! Decode op2 field
2: GOTO 19 CONDITIONED ON ir[24].        ! Goto 19 if Branch format
3: R[rd] ← ir[imm22];                    ! sethi
   GOTO 20.
4: R[15] ← AND(pc, pc).                  ! call: save pc in register 15
5: temp0 ← ADD(ir, ir).                  ! Shift disp30 field left
6: temp0 ← ADD(ir, ir).                  ! Shift again
7: pc ← ADD(pc, temp0); GOTO 0.          ! Jump to subroutine
   ! Get second source operand into temp0 for Arithmetic format
8: temp0 ← { SEXT13(ir) CONDITIONED ON ir[13]×NOR(ir[19:22]), ! addcc
             R[rs2] CONDITIONED ON ir[13]×NOR(ir[19:22]),      ! addcc
             SIMM13(ir) CONDITIONED ON ir[13]×OR(ir[19:22]),   ! Remaining
             R[rs2] CONDITIONED ON ir[13]×OR(ir[19:22])}. ! Arithmetic instructions
   ! Decode op3 field for Arithmetic format
9: R[rd] ← {
   ADDCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010000), ! addcc
   ANDCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010001), ! andcc
   ORCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010010),  ! orcc
   NORCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010110), ! ornc
   SRL(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 100110),   ! srl
   ADD(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 111000)}; ! jmp1
   GOTO 20.
   ! Get second source operand into temp0 for Memory format
10: temp0 ← {SEXT13(ir) CONDITIONED ON ir[13],
             R[rs2] CONDITIONED ON ir[13]}.
11: temp0 ← ADD(R[rs1], temp0).
   ! Decode op3 field for Memory format
   GOTO {12 CONDITIONED ON ir[21],          ! ld
         13 CONDITIONED ON ir[21]}.        ! st
12: R[rd] ← AND(temp0, temp0); Read ← 1; GOTO 20.
13: ir ← RSHIFT5(ir).

```

Figure 6-22 HDL description of the ARC control unit.

have to do additional decoding depending on the value of the `op` field. At line 4, which is for the Call format, no additional decoding is necessary. The `call` instruction is then implemented in statements 4-7, which are similar to the microcoded version.

```

14: ir ← RSHIFT5(ir).
15: ir ← RSHIFT5(ir).
16: ir ← RSHIFT5(ir).
17: ir ← RSHIFT5(ir).
18: r0 ← AND(temp0, R[rs2]); Write ← 1; GOTO 20.
19: pc ← { ! Branch instructions
      ADD(pc, temp0) CONDITIONED ON  $\overline{\text{ir}[28]} + \overline{\text{ir}[28] \times \text{ir}[27] \times \text{Z}} +$ 
       $\overline{\text{ir}[28] \times \text{ir}[27] \times \text{ir}[26] \times \text{C}} + \overline{\text{ir}[28] \times \text{ir}[27] \times \text{ir}[26] \times \text{ir}[25] \times \text{N}} +$ 
       $\overline{\text{ir}[28] \times \text{ir}[27] \times \text{ir}[26] \times \text{ir}[25] \times \text{V}},$ 
      INCPc(pc) CONDITIONED ON  $\overline{\text{ir}[28] \times \text{ir}[27] \times \text{Z}} +$ 
       $\overline{\text{ir}[28] \times \text{ir}[27] \times \text{ir}[26] \times \text{C}} + \overline{\text{ir}[28] \times \text{ir}[27] \times \text{ir}[26] \times \text{ir}[25] \times \text{N}} +$ 
       $\overline{\text{ir}[28] \times \text{ir}[27] \times \text{ir}[26] \times \text{ir}[25] \times \text{V}};$ 
      GOTO 0.
20: pc ← INCPc(pc); GOTO 0.
END SEQUENCE.
END ARC_CONTROL_UNIT.

```

Figure 6-22 (Continued.)

In statement 2, additional decoding is performed on the `op2` field which is checked to determine if the instruction is `sethi` or a branch. Since there are only two possibilities, only one bit of `op2` needs to be checked in line 2. Line 3 then implements `sethi` and line 19 implements the branch instructions.

Line 8 begins the Arithmetic format section of the code. Line 8 gets the second source operand, which can be either immediate or direct, and can be sign extended to 32 bits (for `addcc`) or not sign extended. Line 9 implements the Arithmetic format instructions, conditioned on the `op3` field. The XNOR function returns true if its arguments are equal, otherwise it returns false, which is useful in making comparisons.

Line 10 begins the Memory format section of the code. Line 10 gets the second source operand, which can either be a register or an immediate operand. Line 11 decodes the `op3` field. Since the only Memory format instructions are `ld` and `st`, only a single bit (`IR[21]`) needs to be observed in the `op3` field. Line 12 then implements the `ld` instruction, and lines 13-18 implement the `st` instruction. Finally, line 20 increments the program counter and transfers control back to the first statement.

Now that the control sequence is defined, the next step is to design the logic for the control section. Since there are 21 statements, there are 21 flip-flops in the control section as shown in Figure 6-23. A control signal (CS_i) is produced for each of the 21 states, which is used in the data section of the hardwired controller.

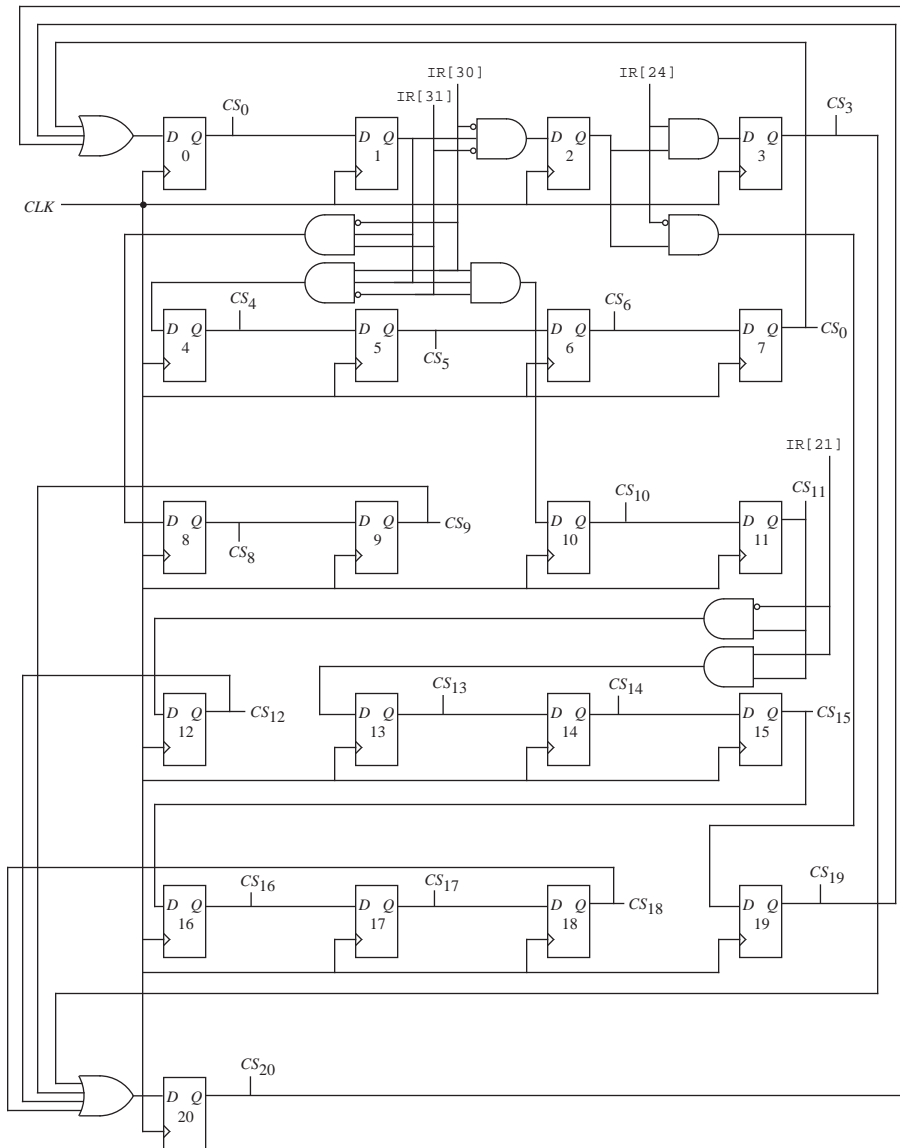


Figure 6-23 The hardwired control section of the ARC: generation of the control signals.

In Figure 6-24, the data section of the hardwired controller generates the signals that control the datapath. There are 27 OR gates that correspond to the 27 signals that control the datapath. (Refer to Figure 6-10. Count the 27 signals that originate in the control section that terminate in the datapath.) The AMUX signal is set to 1 only in lines 9 and 11, which correspond to operations that place

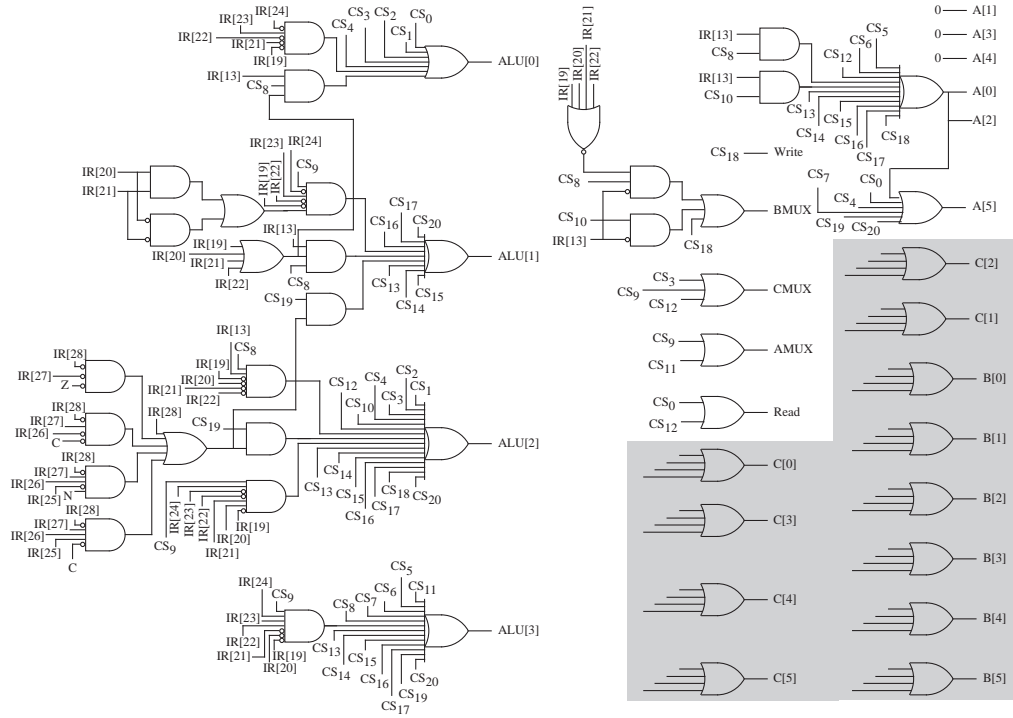


Figure 6-24 The hardwired control section of the ARC: signals from the data section of the control unit to the datapath. (Shaded areas are not detailed.)

$rs1$ onto the A bus. Signals CS_9 and CS_{11} are thus logically OR'd to produce AMUX. Likewise, rd is placed on the C bus in lines 3, 9, and 12, and so CS_3 , CS_9 , and CS_{12} are logically OR'd to produce CMUX.

The BMUX signal is more complex. $rs2$ is placed on the B bus in lines 8, 10, and 18, and so CS_8 , CS_{10} , and CS_{18} are used to generate BMUX as shown. However, in line 8, BMUX is set (indicating $rs2$ is placed on the B bus) only if $IR[13] = 0$ and $IR[19:22]$ are all 0 (for the rightmost 4 bits of the 6-bit $op3$ pattern for $addcc$: 010000.) The corresponding logic is shown for this case. Likewise, in line 10, BMUX is set to 1 only when $IR[13] = 0$. Again, the corresponding logic is shown.

The Read signal is set in lines 10 and 12, and so CS_0 and CS_{12} are logically OR'd to produce Read. The Write signal is generated only in line 18, and thus needs no logic other than the signal CS_{18} .

There are 4 signals that control the ALU: ALU[0], ALU[1], ALU[2], and ALU[3], which correspond to F_0 , F_1 , F_2 , and F_3 , respectively, in the ALU operation table shown in Figure 9-4. These four signals need values in each of the 20 HDL lines. In line 0, the ALU operation is AND, which corresponds to $ALU[3:0] = 0101$. Line 1 has no ALU operation specified, and so we can arbitrarily choose an ALU operation that has no side effects, like AND (0101). Continuing in this way, taking `CONDITIONED ON` statements into account, produces the logic for ALU[3:0] as shown in the figure.

The control signals are sent to the datapath, similar to the way that the MIR controls the datapath in the microprogrammed approach of Figure 6-10. The hardwired and microcontrolled approaches can thus be considered interchangeable, except with varying costs. There are only 21 flip-flops in the hardwired approach, but there are $2048 \times 41 = 83,968$ flip-flops in the microprogrammed approach (although in actuality, a ROM would be used, which consumes less space because smaller storage elements than flip/flops can be used.) The amount of additional combinational logic is comparable. The hardwired approach is faster in executing ARC instructions, especially in decoding the Branch format instructions, but is more difficult to change once it is committed to fabrication.

EXAMPLE

Consider adding the same `subcc` instruction from the previous EXAMPLE to the hardwired implementation of the ARC instruction set. As before, the `subcc` instruction uses the Arithmetic format and an `op3` field of 001100.

Only line 9 of the HDL code needs to be changed, by inserting the expression:

```
ADDCC (R[rs1], INC_1(temp0)) CONDITIONED ON XNOR(IR[19:24], 001100), ! subcc
```

before the line for `addcc`.

The corresponding signals that need to be modified are $ALU[3:0]$. The `INC_1` construct in the line above indicates that an adder CLU, which would be defined in another HDL module, should be created (in a hardwired control unit, there is a lot of flexibility on what can be done.) ■

6.4 Case Study: A User-Microprogrammable Computer: The PDP11/60

Although it was introduced over 20 years ago, in 1977, as an extension to DEC's

minicomputer family, the PDP 11/60 control unit contains several features that still make it interesting as a case study. The PDP11/60 was a state-of-the art minicomputer when introduced, having an 18-bit address space and a 16-bit word size. It is a two-address machine (one of the two source operands is the same as the destination operand) and has 8 16-bit registers, numbered R0 through R7, with R6 serving as a hardware stack pointer (SP) and R7 serving as the PC. It also has an 8-bit Processor Status Register (PSW). The PSW contains N, Z, V, and C bits, as well as a 3-bit interrupt priority field. The PDP 11/60 has an integral floating point processor that executes floating point instructions in parallel with integer instructions. It was one of the first minicomputers to include cache memory, and it has a direct memory access (DMA) unit for high-speed I/O (see Chapter 8 for details on DMA.)[†]

6.4.1 MICROPROGRAMMED vs. HARDWIRED CONTROL UNITS

One of the more interesting features of this machine is its approach to control unit design. At the time of its introduction in June 1977, most minicomputers were microprogrammed, with the microprogram being stored in a separate ROM. The reason for this was primarily one of convenience, as microprograms could be more easily written, debugged, and maintained, because of the software-like nature of microprogramming. Microprogrammed machines had a disadvantage, however: each microstep requires a memory fetch. This means that the microprogrammed approach suffers a performance disadvantage relative to hardwired control units as long as memory access times are longer than propagation time through a few levels of combinational logic.

Hardwired control logic, on the other hand, while having a speed advantage, is difficult to design without effective computer-aided design (CAD) tools, which were largely lacking at the time. It also results in a design that is harder to modify and upgrade, as compared with a ROM-based microprogram, which can be upgraded by plugging in a new ROM.

6.4.2 STATISTICS IN COMPUTER DESIGN

The designers of the 11/60 decided to use the hardwired approach for instructions (or parts of instructions) that were frequently used, and microprogramming for the rest of the instruction set. By analyzing the frequency of use of instruc-

†. Much of the material in this case study was taken from (Mudge, 1978).

tions, the designers were able to determine where hardwiring would be most advantageous, and implemented those instructions with a hardwired approach. The machine approached the speed made possible with a hardwired control unit, while retaining advantages of microprogramming. Over one million floating point instructions from actual user programs were analyzed to guide the trade-off, which was a large sample at the time.

6.4.3 THE PDP 11/60 CONTROL UNIT

The control unit of the PDP 11 family is fairly simple. Figure 6-25 shows the

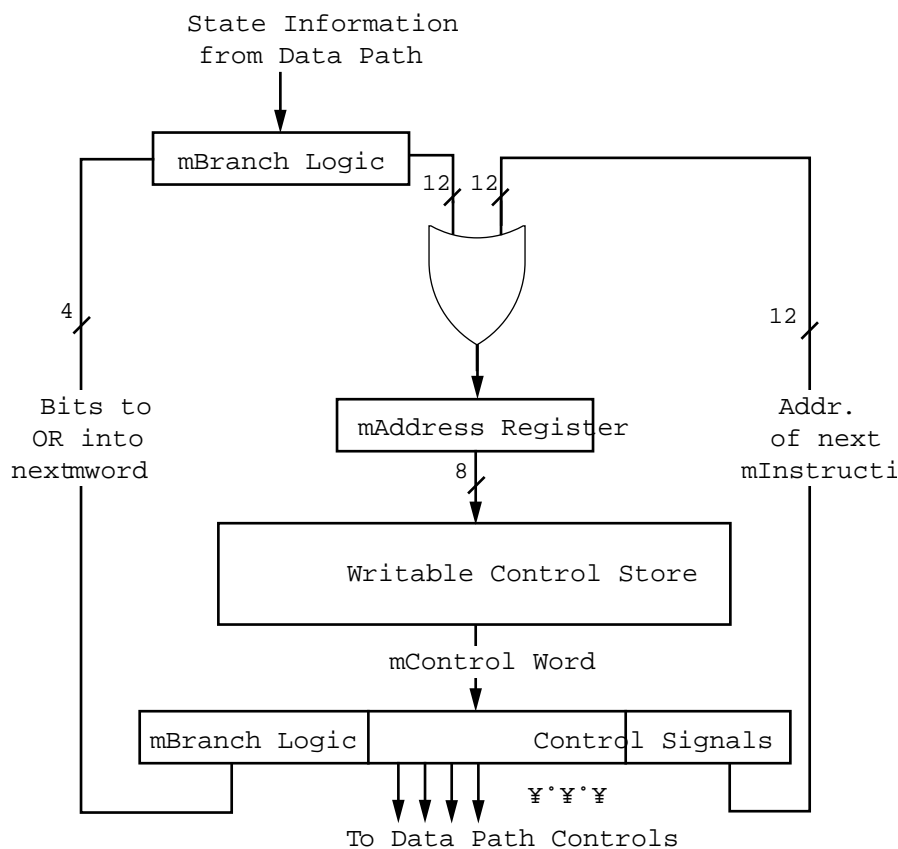


Figure 6-25 The PDP 11 control unit.

design of a PDP 11 microcode control unit. Microaddresses are 12 bits wide, allowing the addressing of 4096 microcontrol words. Notice that each microinstruction contains, in addition to the data path control signals, two additional

fields that select the address of the next microinstruction. The 12-bit next address field effectively makes each microinstruction a branch instruction. This feature is feasible in microcontrol units, where addresses are small, and it allows great flexibility in microcode design. The other field, microbranch logic, is a 4-bit field whose bits are combined with bits in the data path such as the flags, in the block marked, “microBranch Logic.” The resulting 10-bit word is ORed with the next address bits. This “bit ORing” allows additional branch capabilities, such as the ability to branch to a certain microaddress based upon the value of a flag in the processor status register.

6.4.4 WRITABLE CONTROL STORE AND USER MICROPROGRAMMING

One of the innovations of the 11/60 was its use of a Writable Control Store (WCS) to store microinstructions, rather than the usual ROM storage. The WCS had a 4096-word address space for the microprogram that was loaded from disk. This meant that the microcode could be easily updated with bug fixes (yes, there can be bugs in microcode) or instruction set enhancements and upgrades.

Furthermore, the top 1024 words were reserved for user microprograms. DEC made several provisions to support user microprogramming, including the incorporation of an opcode specifically reserved for user microinstructions. The designers also included development tools and documentation aimed at assisting the end user with microprogram development. They even provided a means for users to write microprogrammed interrupt service routines.

While this was an interesting and innovative approach, it was abandoned in later family members, perhaps because of the increased difficulty in assuring upward compatibility of user programs containing user-developed instructions.

■ SUMMARY

A microarchitecture consists of a datapath and a control section. The datapath contains data registers, an ALU, and the connections among them. The control section contains registers for microinstructions (for a microprogramming approach) and for condition codes, and a controller. The controller can be microprogrammed or hardwired. A microprogrammed controller interprets microinstructions by executing a microprogram that is stored in a control store. A hardwired controller is organized as a collection of flip-flops that maintain state

information, and combinational logic that implements transitions among the states.

The hardwired approach is fast, and consumes a small amount of hardware in comparison with the microprogrammed approach. The microprogrammed approach is flexible, and simplifies the process of modifying the instruction set. The control store consumes a significant amount of hardware, which can be reduced to a degree through the use of nanoprogramming. Nanoprogramming adds delay to the microinstruction execution time. The choice of microprogrammed or hardwired control thus involves trade-offs: the microprogrammed approach is large and slow, but is flexible and lends itself to simple implementations, whereas the hardwired approach is small and fast, but is difficult to modify, and typically results in more complicated implementations.

■ FURTHER READING

(Wilkes, 1958) is a classic reference on microprogramming. (Mudge, 1978) covers microprogramming on the DEC PDP 11/60. (Tanenbaum, 1990) and (Mano, 1991) provide instructional examples of microprogrammed architectures. (Hill and Peterson, 1987) gives a tutorial treatment of the AHPL hardware description language, and hardwired control in general. (Lipsett et al., 1989) and (Navabi, 1993) describe the commercial VHDL hardware description language and provide examples of its use. (Gajski, 1988) covers various aspects of silicon compilation.

Gajski, D., *Silicon Compilation*, Addison Wesley, (1988).

Hill, F. J. and G. R. Peterson, *Digital Systems: Hardware Organization and Design*, 3/e, John Wiley & Sons, (1987).

Lipsett, R., C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, (1989).

Mano, M., *Digital Design*, 2/e, Prentice Hall, (1991).

Mudge, J. Craig, *Design Decisions for the PDP11/60 Mid-Range Minicomputer*, in *Computer Engineering, A DEC View of Hardware Systems Design*, Digital Press,

Bedford MA, (1978).

Navabi, Z., *VHDL: Analysis and Modeling of Digital Systems*, McGraw Hill, (1993).

Tanenbaum, A., *Structured Computer Organization*, 3/e, Prentice Hall, Englewood Cliffs, New Jersey, (1990).

Wilkes, M. V., W. Redwick, and D. Wheeler, "The Design of a Control Unit of an Electronic Digital Computer," *Proc. IRE*, vol. 105, p. 21, (1958).

PROBLEMS

6.1 Design a 1-bit arithmetic logic unit (ALU) using the circuit shown in Figure 6-26 that performs bitwise addition, AND, OR, and NOT on the 1-bit

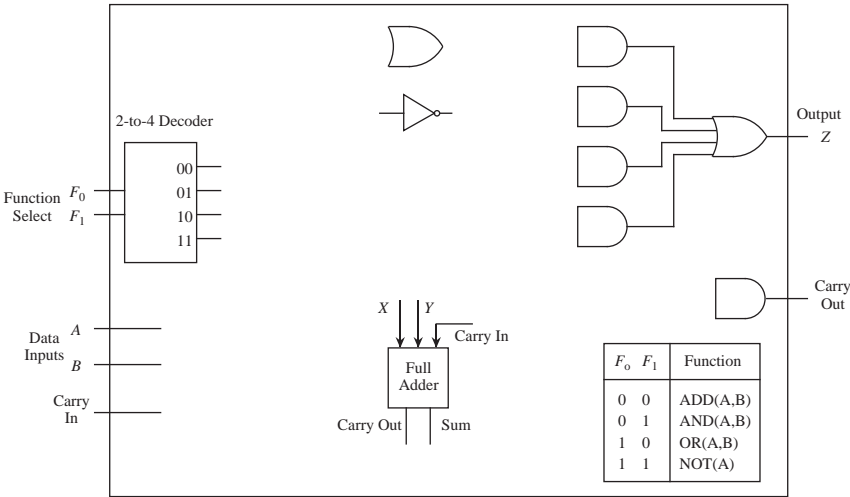


Figure 6-26 A one-bit ALU.

inputs A and B . A 1-bit output Z is produced for each operation, and a carry is also produced for the case of addition. The carry is zero for AND, OR, and NOT. Design the 1-bit ALU using the components shown in the diagram. Just draw the connections among the components. Do not add any logic gates, MUXes, or anything else. Note: The Full Adder takes two one-bit inputs (X and Y) and a Carry In, and produces a Sum and a Carry Out.

6.2 Design an ALU that takes two 8-bit operands X and Y and produces an

8-bit output Z . There is also a two-bit control input C in which 00 selects logical AND, 01 selects OR, 10 selects NOR, and 11 selects XOR. In designing your ALU, follow this procedure: (1) draw a block diagram of eight 1-bit ALUs that each accept a single bit from X and Y and both control bits, and produce the corresponding single-bit output for Z ; (2) create a truth table that describes a 1-bit ALU; (3) design one of the 1-bit ALUs using an 8-to-1 MUX.

6.3 Design a control unit for a simple hand-held video game in which a character on the display catches objects. Treat this as an FSM problem, in which you only show the state transition diagram. Do not show a circuit. The input to the control unit is a two-bit vector in which 00 means “Move Left,” 01 means “Move Right,” 10 means “Do Not Move,” and 11 means “Halt.” The output Z is 11 if the machine is halted, and is 00, 01, or 10 otherwise, corresponding to the input patterns. Once the machine is halted, it must remain in the halted state indefinitely.

6.4 In Figure 6-3, there is no line from the output of the C Decoder to $\%r0$. Why is this the case?

6.5 Refer to diagram Figure 6-27. Registers 0, 1, and 2 are general purpose registers. Register 3 is initialized to the value +1, which can be changed by the microcode, but you must make certain that it does not get changed.

a) Write a control sequence that forms the two’s complement difference of the contents of registers 0 and 1, leaving the result in register 0. Symbolically, this might be written as: $r0 \leftarrow r0 - r1$. Do not change any registers except $r0$ and $r1$ (if needed). Fill in the table shown below with 0’s or 1’s (use 0’s when the choice of 0 or 1 does not matter) as appropriate. Assume that when no registers are selected for the A-bus or the B-bus, that the bus takes on a value of 0.

Write Enables				A-bus enables				B-bus enables				$F_0 F_1$		Time
0	1	2	3	0	1	2	3	0	1	2	3	F_0	F_1	
														0
														1
														2

b) Write a control sequence that forms the exclusive-OR of the contents of

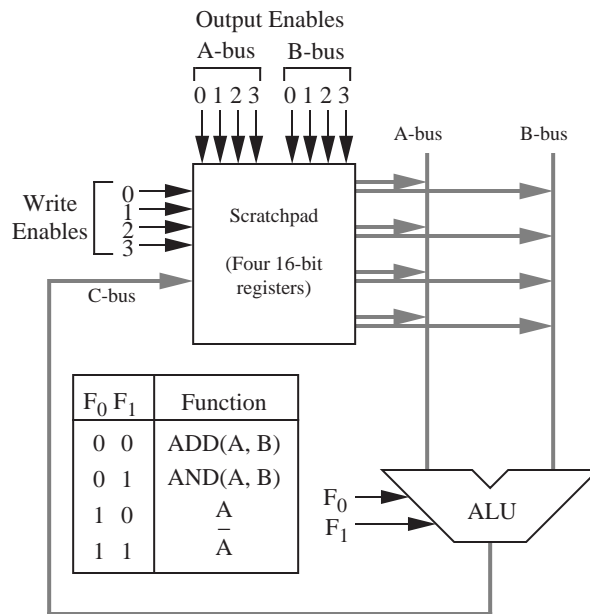


Figure 6-27 A small microarchitecture.

registers 0 and 1, leaving the result in register 0. Symbolically, this might be written as: $r0 \leftarrow \text{XOR}(r0, r1)$. Use the same style of solution as for part (a).

6.6 Write the binary form for the microinstructions shown below. Use the style shown in Figure 6-17. Use the value 0 for any fields that are not needed.

60: $R[\text{temp0}] \leftarrow \text{NOR}(R[0], R[\text{temp0}]); \text{ IF } Z \text{ THEN GOTO } 64;$
 61: $R[\text{rd}] \leftarrow \text{INC}(R[\text{rs1}]);$

6.7 Three binary words are shown below, each of which can be interpreted as a microinstruction. Write the mnemonic version of the binary words using the micro-assembly language introduced in this chapter.

A		B		C							
M		M		M							
U		U		U		URW					
A	X	B	X	C	X	D R	ALU	COND	JUMP	ADDR	
1 0 0 1 0 1	0	0 0 0 0 0 0	0	1 0 0 0 0 1	0	0 0	1 1 0 0	0 0 0	0 0 0 0 0 0 0 0 0 0	0	
0 0 0 0 0 0	1	1 0 0 0 0 1	0	1 0 0 0 0 1	0	0 0	1 0 0 0	1 1 0	1 1 1 0 0 0 0 0 0 1	1	
0 0 0 0 0 0	1	0 0 0 0 0 0	1	1 0 0 0 0 1	0	0 0	1 0 0 0	1 0 1	1 1 1 0 0 0 1 0 0 1	0	

- 6.8** Rewrite the microcode for the `call` instruction starting at line 1280 so that only 3 lines of microcode are used instead of 4. Use the `LSHIFT2` operation once instead of using `ADD` twice.
- 6.9** (a) How many microinstructions are executed in interpreting the `subcc` instruction that was introduced in the first Example section? Write the numbers of the microinstructions in the order they are executed, starting with microinstruction 0.
- (b) Using the hardwired approach for the ARC microcontroller, how many states are visited in interpreting the `addcc` instruction? Write the states in the order they are executed, starting with state 0.
- 6.10** (a) List the microinstructions that are executed in interpreting the `ba` instruction.
- (b) List the states (Figure 6-22) that are visited in interpreting the `ba` instruction.
- 6.11** Register `%r0` can be designed using only tri-state buffers. Show this design.
- 6.12** What bit pattern should be placed in the `C` field of a microword if none of the registers are to be changed?
- 6.13** A control unit for a machine tool is shown in Figure 6-28. You are to create the microcode for this machine. The behavior of the machine is as follows: If the Halt input *A* is ever set to 1, then the output of the machine stays halted forever and outputs a perpetual 1 on the *X*line, and 0 on the *V* and *W*lines. A waiting light (output *V*) is enabled (set to 1) when no inputs are enabled. That is, *V* is lit when the *A*, *B*, and *C* inputs are 0, and the machine is not halted. A bell is sounded (*W*=1) on every input event (*B*=1 and/or *C*=1) except when the machine is halted. Input *D* and output *S* can be used for state information for your microcode. Use 0's for any fields that do not matter. Hint: Fill in the lower half of the table first.
- 6.14** For this problem, you are to extend the ARC instruction set to include a new instruction by modifying the microprogram. The new ARC instruction

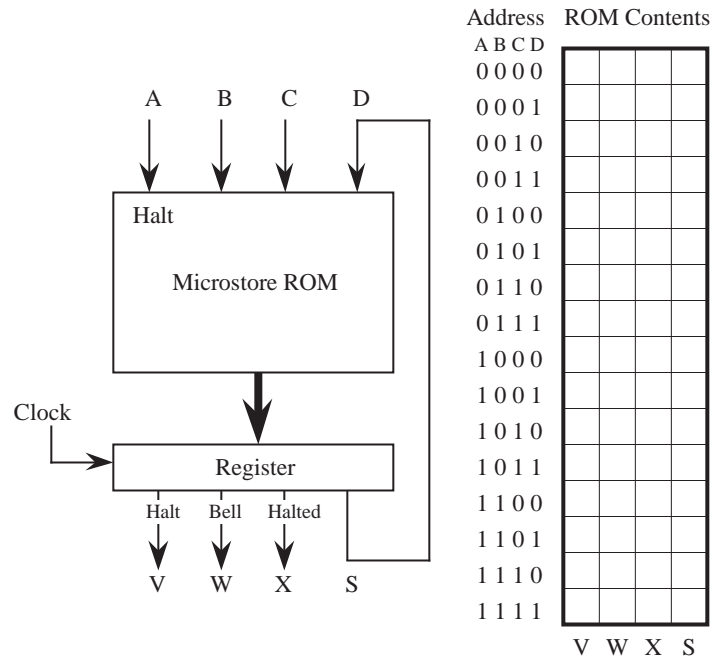


Figure 6-28 Control unit for a machine tool.

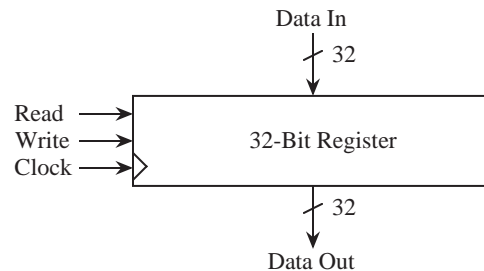
to be microcoded is:

`xorcc` — Perform an exclusive OR on the operands, and set the condition codes accordingly. This is an Arithmetic format instruction. The `op3` field is 010011.

Show the new microinstructions that will be added for `xorcc`.

6.15 Show a design for a four-word register stack, using 32-bit registers of the

form shown below:



Four registers are stacked so that the output of the top register is the input to the second register, which outputs to the input of the third, which outputs to the input of the fourth. The input to the stack goes into the top register, and the output of the stack is taken from the output of the top register (*not* the bottom register). There are two additional control lines, `push` and `pop`, which cause data to be pushed onto the stack or popped off the stack, respectively, when the corresponding line is 1. If neither line is 1, or if both lines are 1, then the stack is unchanged.

6.16 In line 1792 of the ARC microprogram, the conditional `GOTO` appears at the end of the line, but in line 8 it appears at the beginning. Does the position of the `GOTO` within a micro-assembly line matter?

6.17 A microarchitecture is shown in Figure 6-29. The datapath has four registers and an ALU. The control section is a finite state machine, in which there is a RAM and a register. For this microarchitecture, a compiler translates a high level program directly into microcode; there is no intermediate assembly language form, and so there are no instruction fetch or decode cycles.

For this problem, you are to write the microcode that implements the instructions listed below. The microcode should be stored in locations 0, 1, 2, and 3 of the RAM. Although there are no lines that show it, assume that the *n* and *z* bits are both 0 when $C_0C_1 = 00$. That is, A_{23} and A_{22} are both 0 when there is no possible jump. Note: Each bit of the A, B, and C fields corresponds directly to a register. Thus, the pattern 1000 selects register R3, *not* register 8, which does not exist. There are some complexities with respect to how branches are made in this microarchitecture, but you do not need to be concerned with how this is done in order to generate the microcode.

3: Jump to $(20)_{10}$

6.18 In line 2047 of the ARC microprogram shown in Figure 6-15, would the program behave differently if the “GOTO 0” portion of the instruction is deleted?

6.19 In **horizontal microprogramming**, the microwords are wide, whereas in **vertical microprogramming** the words are narrow. In general, horizontal microwords can be executed quickly, but require more space than vertical microwords, which take more time to execute. If we make the microword format shown in Figure 6-11 more horizontal by expanding the A, B, and C fields to contain a single bit for each of the 38 registers instead of a coded six-bit version, then we can eliminate the A, B, and C decoders shown in Figure 6-3. This allows the clock frequency to be increased, but also increases the space for the microstore.

(a) How wide will the new horizontal microword be?

(b) By what percentage will the microstore increase in size?

6.20 Refer to Figure 6-7. Show the ALU LUT_0 and ALU LUT_x ($x > 0$) entries for the INC(A) operation.

6.21 On some architectures, there is special hardware that updates the PC, which takes into account the fact that the rightmost two bits are always 0. This is not shown in this chapter for the ARC, and so the branch microcode in lines 2 - 20 of Figure 6-15 has an error in how the PC is updated on line 12. Identify the error, and show the corrected microcode.

