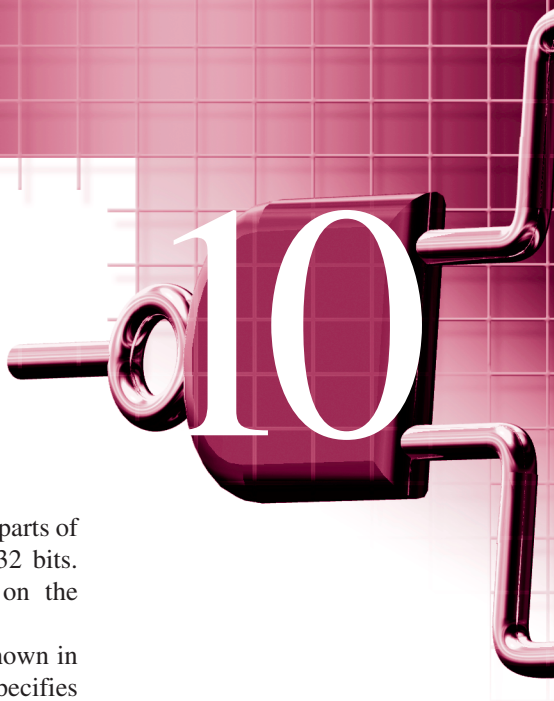
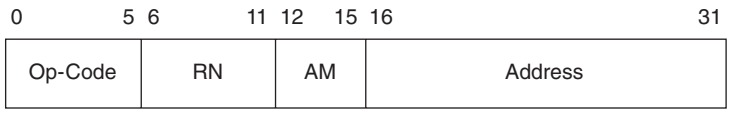
The Design of a Central Processing Unit



In this chapter, we will discuss the design of the controller for parts of a computer, called MODEL. MODEL has a word size of 32 bits. Instructions may require one or two words, depending on the addressing mode.

The format for the first word of a MODEL instruction is shown in Figure 10.1, where RN specifies the register number and AM specifies the address mode.

Figure 10.1 Instruction format for MODEL.



10.1 DESCRIPTION OF MODEL

In this section, we will specify the addressing modes and the instructions for which we will show the control sequence and discuss the timing. MODEL would surely have a wider variety of instructions and more addressing modes. However, those that we specify will be adequate to demonstrate the process of design.

10.1.1 Memory and Register Set

MODEL has a memory space of 2^{32} words,* each 32 bits wide. To access memory, the address is placed on lines `AD[0:31]` for one clock period. For read, a 1 is placed on line **`read`** at that same time, and the contents

*This implies a 32-bit address and a maximum memory of 2^{32} words. Not all of the memory needs to be there for the system to work properly.

of that memory location will be available on bus DATA[0:31] during that clock period. Thus, a typical memory fetch step might be

```
AD = PC; read = 1; IR ← DATA.
```

To store in memory, the word is connected to DATA (at the same time as the address is on AD) and a 1 is put on line **write**, for example

```
AD = PC; DATA = WORK; write = 1.
```

We will examine the modifications needed to handle a slower memory in Section 10.4.

DATA is an INTERSYSTEM BUS. ADIN, **read**, and **write** could be thought of as OUTPUT LINES from MODEL or as an INTERSYSTEM BUS. We will treat them as the latter; they are part of BUS, as shown in Figure 8.1.

The register set of MODEL includes the following registers:

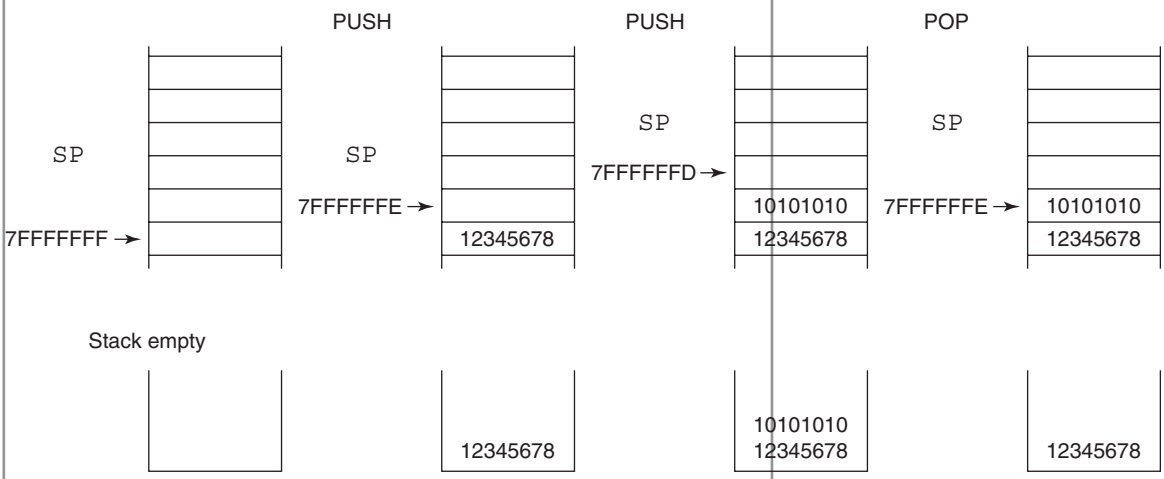
PC[0:31]	The program counter
IR[0:31]§	The instruction register—a place to store the first word of an instruction while it is being decoded and executed
REG[0:63; 0:31]*	A set of 64 general-purpose registers
WORK[0:31]§	A register to hold data temporarily
EA[0:31]§	Register in which the effective address is computed
z	Zero flag bit—set to 1 when the result of some instructions is zero† (0 otherwise)
n	Negative flag bit—set to 1 when the result of some instructions is negative (leading bit is 1)
c	Carry (and borrow) bit—stores the carry out of the most significant bit of the adder
v	Two's complement overflow bit—set to 1 if the result of an operation is out of range, assuming operands are in two's complement notation (0 otherwise)

Registers indicated with a § contain no useful information between instructions. If we expand the instruction set, we may need to add some registers.

*REG^{3F} will be used as the stack pointer; we will use the notation SP in the DDL, but will define it as REG^{3F} with a NAME definition line.

†When we describe the instructions in Section 10.1.2, we will specify which instructions modify which flag bits.

Figure 10.2 Operation of the stack pointer.



The stack in MODEL is stored in memory. The register SP points to the next empty place on the stack. Elements are stored in descending order on the stack. Thus, if the stack pointer contains 7FFFFFFF and something is pushed onto the stack, it is stored in location 7FFFFFFF and the stack pointer is then decremented to 7FFFFFFE. Figure 10.2 shows the behavior of the stack with two items being pushed onto the stack and then one popped from the stack. Note that the SP is decremented on pushes and incremented on pops. When something is popped, it is not erased; it is copied to the Central Processing Unit (CPU) and the pointer is incremented. After the pop, the contents of 7FFFFFFE are still there but will never be used by a stack instruction. A push would write over it; a pop would first increment SP and take the contents of 7FFFFFFF.

Internally, data is transferred by way of a 32-bit internal bus, CPUBUS. In addition, the arithmetic and logic unit has two 32-bit input buses: INA and INB. In describing the behavior of the machine, these buses are not referenced most of the time. A statement

```
REG/IR6:11 ← WORK.
```

implies that WORK is connected to CPUBUS and the data on that bus is clocked into the register, that is,

```
CPUBUS = WORK; REG/IR6:11 ← CPUBUS.
```

EXAMPLE 10.1

```
WORK ← ADD1:32[FFFFFFFF; WORK; 0]
```

implies the constant FFFFFFFF is connected to one 32-bit input of the adder, WORK is connected to the other 32-bit input, 0 is connected to the

c$_{in}$ input, the right 32 bits of the adder output is connected to CPUBUS, and the bus is clocked into WORK. In this example, the carry output of the adder is not stored anywhere,

```
INA = FFFFFFFF; INB = WORK; c_in = 0;
BUS = ADD_1:32[INA; INB; c_in]; WORK ← CPUBUS
```

If we wanted to store that in the **c** flip flop, we would have written

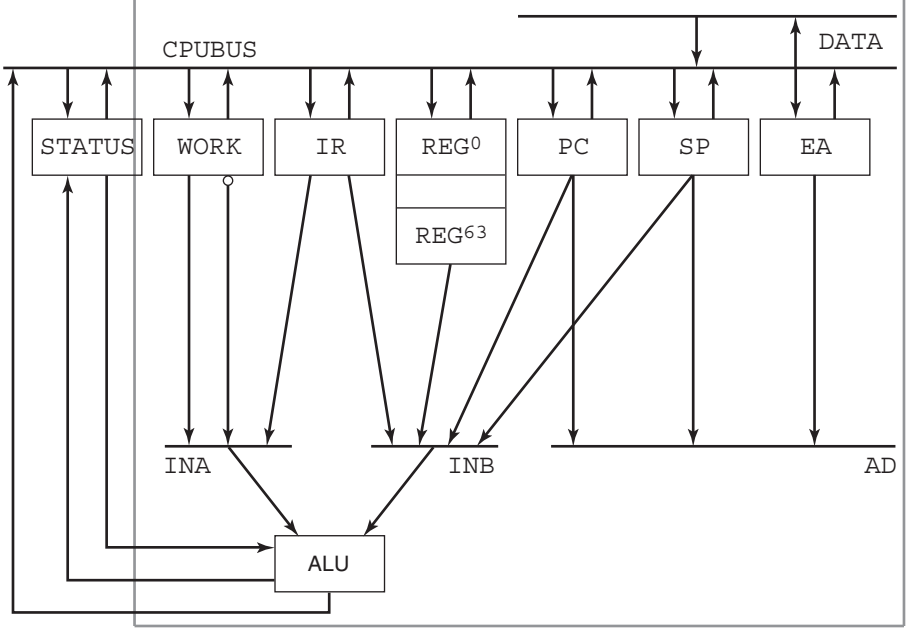
```
c, WORK ← ADD[FFFFFFFF; WORK; 0]
```

Since there is no other way to move data, it is not necessary to be more specific.

Figure 10.3 shows a simplified block diagram of the bus structure. The constants, partial register connections, and shifted WORK are not shown, (For example, WORK ← FFFF, IR$_{16:31}$ implies the constant FFFF is connected to the left half of CPUBUS and only the right half of IR is connected to the right half of CPUBUS.)

In addition to the three internal busses, there are two intersystem busses: DATA and AD. (The bus signals **read** and **write** are not shown.) Note that INA, INB, AD, **read** and **write** are really only multiplexors, with data only going in one direction. (However, in a larger

Figure 10.3 MODEL bus diagram.



system where memory is used by more than one subsystem, AD, **read**, and write may be buses.)

10.1.2 Addressing Modes

We will define 7 addressing modes (of the 16 possibilities with 4 bits), treating the remaining 9 as no-ops (no operation). Some of the Examples, Solved Problems, and Exercises will suggest others. In each of the following examples, we will show what happens for a load register instruction (LOD), where RN is assumed to be 5 (05). The first four modes require a one-word instruction; the others require a second word. The AM field is shown in binary, rather than hexadecimal, to simplify the discussion later.

Register (AM = 0000) The data comes from or is stored in the one of the 64 registers specified by $IR_{26:31}$.* This mode is not valid for branch instructions, since they require a memory address.

Example: LOD REG^5, REG^{14} $[IR_{26:31} = OE]$
The data in Register 14 is loaded into Register 5.

Register indirect (AM = 0010) The register specified by $IR_{26:31}$ contains the address in main memory of the data or where the result is to be stored or the jump is to go.

Example: LOD REG^5, (REG^{14}) $[IR_{26:31} = OE]$
where REG^{14}: 12345678
The data in memory location 12345678 is loaded into Register 5.

Page zero (AM = 0110) Bits $IR_{16:31}$ are zero-extended to produce the effective address.

Example: LOD REG^5, z1234 $[IR_{16:31} = 1234]$
The data in memory location 00001234 is loaded into Register 5.

Relative (AM = 0111) Bits $IR_{16:31}$ are sign-extended and added to the program counter (after the program counter has been incremented to point to the next instruction) to produce the effective address.

Example: LOD REG^5, @1234 $[IR_{16:31} = 1234]$
If this instruction is at address 01120111, the effective address is

$$00001234 + 01120111 + 1 = 01121346.$$

The data in memory location 01121346 is loaded into Register 5.

*In those modes where the address field specifies a register, $IR_{16:25}$ (the rest of the address field) are ignored.

Example: LOD REG[5], @8000 [$IR_{16:31}$ = 8000]
If this instruction is at address 01120111, the effective address is

FFFF8000 + 01120111 + 1 = 01118112.

The data in memory location 01118112 is loaded into Register 5.

Direct (AM = 1000) [$IR_{16:31}$ are ignored] A second word of the instruction is required. That word contains the effective address.

Example: LOD REG[5], 12345678
The contents of memory location 12345678 is loaded into Register 5.

Indirect (AM = 1001) [$IR_{16:31}$ are ignored] A second word of the instruction is required. It contains the address in memory where the effective address of data is found.

Example: LOD REG[5], (12345678)

M[12345678] = 56789ABC

The contents of 12345678 are fetched. Then the contents of memory location 56789ABC are loaded into Register 5.

Immediate (AM = 1100) [$IR_{16:31}$ are ignored] A second word of the instruction is required. It contains the data. This type is not valid for any instruction that requires an address (for storing a result or branching).

Example: LOD REG[5], #12345678
The number 12345678 is loaded into Register 5. (The constant 12345678 is contained in the second word of the instruction.)

EXAMPLE 10.2

We have an instruction to ADD the number specified by the addressing to REG[4], where

This instruction is at location 12341234
The second word of the instruction (if any) is 20000000
Bits 16 to 31 of this instruction word are AB07
REG[4] = 00000102
REG[7] = 00123344
M[0000AB07] = 11111111
M[00123344] = FFFFFFFE
M[1233BD3C] = 44332211
M[20000000] = 00123344

We will examine which registers (not including the flag bits) are changed for each addressing type.

Register REG[4] ← 00123446 (00000102 + 00123344)
PC ← 12341235

Register indirect	$REG^4 \leftarrow$ 00000100	(00000102 + FFFFFFFE)
	PC ← 12341235	
Page zero	$REG^4 \leftarrow$ 11111213	(00000102 + 11111111)
	PC ← 12341235	
Relative	Address = 12341234 + 1 + FFFFAB07 = 1233BD3C	
	$REG^4 \leftarrow$ 44332313	(00000102 + 44332211)
	PC ← 12341235	
Direct	$REG^4 \leftarrow$ 00123446	(00000102 + 00123344)
	PC ← 12341236	
Indirect	$REG^4 \leftarrow$ 00000100	(00000102 + FFFFFFFE)
	PC ← 12341236	
Immediate	$REG^4 \leftarrow$ 20000102	(00000102 + 20000000)
	PC ← 12341236	

[SP 1, 2; EX 1]

10.1.3 Instruction Set of MODEL

In this section, we will define a subset of the instructions, enough to illustrate the design of the controller. For each, we will specify a three-letter mnemonic and the flag bits that are affected. We will not assign an op-code; rather, we will implement the controller, assuming that an appropriate decoder is included. However, those instructions that do not use the address portion (for example, Return from subroutine) begin with a 1; others begin with a 0. For each of the examples, we will assume that we used direct addressing and that

The effective address is 12345678
M[12345678] = 10101234
REG^3 = FFFFFFF8
SP = 7FFFFFF6
M[7FFFFF7] = 98765432

Data Movement*

LOD *z n* Load register with data specified by the address field†

Example: LOD REG^3, 12345678

$REG^3 \leftarrow$ 10101234 $z \leftarrow 0$ $n \leftarrow 0$

STO Store register in location specified by the address field (either memory or a register)

*We will use REG^{63} for the stack pointer and thus will not need special instructions to load or store SP.

†Either the contents of the memory location specified by EA, or the data for immediate or register addressing.

Example: STO REG^3, 12345678

M[12345678] ← FFFFFFF8

PSH Push register onto the stack

Example: PSH REG^3

M[7FFFFFF6] ← FFFFFFF8 SP ← 7FFFFFF5

PSH and POP use only a register; the address field is ignored.

POP *z n* Pop top of stack to register

Example: POP REG^3

SP ← 7FFFFFF7 REG^3 ← 98765432 *z* ← 0 *n* ← 1

Arithmetic Instructions

ADD *z n c v* Add number specified by the address field to the register

Example: ADD REG^3, 12345678

```
  FFFFFFF8
+ 10101234
(1) 1010122C
```

REG^3 ← 1010122C *z* ← 0 *n* ← 0 *c* ← 1 *v* ← 0

ADC *z n c v* Add with carry; add number specified by the address field to the register and the *c* flag

Example: ADC REG^3, 12345678 (*c* was 1)

REG^3 ← 1010122D *z* ← 0 *n* ← 0 *c* ← 1 *v* ← 0

Example: ADC REG^3, 12345678 (*c* was 0)

REG^3 ← 1010122C *z* ← 0 *n* ← 0 *c* ← 1 *v* ← 0

SUB *z n c v* Subtract number specified by the address field from the register

Example: SUB REG^3, 12345678

```
    1
  FFFFFFF8
- EFEFEDCB
(1) EFEFEDC4
```

REG^3 ← EFEFEDC4 *z* ← 0 *n* ← 1 *c* ← 1 *v* ← 0

CMP *z n c v* Compare; behaves the same as subtract, but does not store the difference back to the register. (It is used to compare the same number with several words from memory without having to reload the register.)

Example: CMP REG[3], 12345678

$z \leftarrow 0 \quad n \leftarrow 1 \quad c \leftarrow 1 \quad v \leftarrow 0$

INC *z n* Increments number specified by the address; ignores RN

Example: INC 12345678

M[12345678] ← 10101235 $z \leftarrow 0$ $n \leftarrow 0$

Logic, Shift, and Rotate Instructions

NOT *z* Bit-by-bit complement; ignores RN

Example: NOT 12345678

M[12345678] ← EFEFEDCB $z \leftarrow 0$

AND *z n* Bit-by-bit AND of register with number specified by the address

Example: AND REG[3], 12345678

	1111	1111	1111	1111	1111	1111	1111	1000
AND	0001	0000	0001	0000	0001	0010	0011	0100
	0001	0000	0001	0000	0001	0010	0011	0000

REG[3] ← 1010230 $z \leftarrow 0$ $n \leftarrow 0$

ASR *z* Arithmetic shift right of number specified by the address; number of places specified by the right 5 bits of RN ($IR_{7:11}$)*

Example: ASR 3, 12345678

M[12345678] ← 02020246 $z \leftarrow 0$

ROR Rotate right number specified by the address; number of places specified by the right 5 bits of RN

Example: ROR 3, 12345678

M[12345678] ← 82020246

Branch Instructions

JMP† Jump to the address if the condition specified by the right 4 bits of RN ($IR_{8:11}$) is met; otherwise, continue to the next step. Branch conditions are specified in Table 10.1.

CLL Call subroutine, save return address on the stack. Branch conditions are specified in Table 10.1.

RTS Return from subroutine (unconditional). Pop address from stack.

*The right 5 bits of the RN field is treated as a number between 0 and 31, not as a register reference.

†For conditional Jumps and Calls, the condition is specified by a hexadecimal digit, such as JM2 for Jump if *n* is 1 or CL7 for Call if $v = 0$.

Table 10.1 Branch conditions.

$RN_{8:11}$	*Condition*	$RN_{8:11}$	*Condition*
0000	z	1000	not used
0001	z'	1001	not used
0010	n	1010	not used
0011	n'	1011	not used
0100	c	1100	not used
0101	c'	1101	not used
0110	v	1110	not used
0111	v'	1111	always

EXAMPLE 10.3

We will look at instructions referencing register 4 (REG^4), using Page zero addressing, where

Bits 16 to 31 of this instruction word are AB07
PC = 12341234
REG^4 = 00000102
REG^{63} (SP) = 7FFFFFF5
M[0000AB07] = 81111111
M[7FFFFFF6] = 00000000

We will examine which registers and memory locations are changed for each instruction type. Unless indicated otherwise, the PC is incremented to 12341235.

LOD	$REG^4 \leftarrow$ 81111111 $z \leftarrow 0$ $n \leftarrow 1$
STO	M[0000AB07] $\leftarrow$ 00000102
PSH	M[7FFFFFF5] $\leftarrow$ 00000102 REG^{63}(SP) $\leftarrow$ 7FFFFFF4
POP	REG^{63} (SP) $\leftarrow$ 7FFFFFF6 $REG^4 \leftarrow$ 00000000
	$z \leftarrow 1$ $n \leftarrow 1$
ADD	$REG^4 \leftarrow$ 81111213 $z \leftarrow 0$ $n \leftarrow 1$ $c \leftarrow 0$ $v \leftarrow 0$
ADC	For $c = 0$, same as ADD, for $c = 1$
	$REG^4 \leftarrow$ 81111214 $z \leftarrow 0$ $n \leftarrow 1$ $c \leftarrow 0$ $v \leftarrow 0$
SUB	$REG^4 \leftarrow$ 7EEEEFF1 $z \leftarrow 0$ $n \leftarrow 0$ $c \leftarrow 0$ $v \leftarrow 0$
CMP	$z \leftarrow 0$ $n \leftarrow 0$ $c \leftarrow 0$ $v \leftarrow 0$
INC	M[0000AB07] $\leftarrow$ 81111112 $z \leftarrow 0$ $n \leftarrow 1$
NOT	M[0000AB07] $\leftarrow$ 7EEEEEEE $z \leftarrow 0$
AND	$REG^4 \leftarrow$ 00000100 $z \leftarrow 0$ $n \leftarrow 0$
ASR	M[0000AB07] $\leftarrow$ F8111111 $z \leftarrow 0$ (4 places)
ROR	M[0000AB07] $\leftarrow$ 18111111
JMP	PC $\leftarrow$ 0000AB07
CLL	PC $\leftarrow$ 0000AB07 M[7FFFFFF5] $\leftarrow$ 12341235
	REG^{63} (SP) $\leftarrow$ 7FFFFFF4
RTS	REG^{63} (SP) $\leftarrow$ 7FFFFFF6 PC $\leftarrow$ 00000000

[SP 3, 4; EX 2, 3]

10.2 CONTROL SEQUENCE FOR MODEL*

In this section, we will develop a straightforward control sequence to implement that part of MODEL described in the previous two sections. In the next sections, we will look at its implementation with a hard-wired controller and a microprogrammed controller.

The first word of an instruction is read into the Instruction Register (IR) in the first step of the control sequence. Next, at step 2, the program counter is incremented to point to the next word (either the second word of this instruction or, if this is a one-word instruction, the first word of the next instruction). Throughout the design of the sequencer, we will use INC to represent an incrementer and DEC to represent a decrementer. In practice, there may be such a device as part of the ALU, or these may be implemented using the adder, putting a 1 or −1 on one of the inputs. Also, at step 2, we branch to step 60, the instruction decode step for those instructions where no address is required, or continue to step 3 for address computation.

```
1. AD = PC; read = 1; IR ← DATA.
2. PC ← INC[PC];
   next: 60 (IR0), 3. (else).
```

The addressing mode of all one-word instructions begins with a 0, and that for two-word instructions begins with a 1. At step 3, we separate these.

```
3. next: 12 (IR12), 4 (else)
4. next/IR13:15: 5, 1, 6, 1, 1, 1, 10, 11.
```

Unused codes are treated as no-ops, branching back to step 1 to fetch a new instruction. When addressing is completed, the address (if there is one) is stored in EA; control then goes to step 18 to fetch data. Those addressing modes that produce data, but no address (immediate and register), store that data in WORK and branch to step 20.

For register and register indirect addressing, bits 26 to 31 specify the register. Thus, at step 5, that register is moved to WORK (for register addressing), and control goes next to step 20. At steps 6 (for register indirect), that register is moved to EA, with control going to step 18.

Register

```
5. WORK ← REG/IR26:31;
   next: 20.
```

Register Indirect

```
6. EA ← REG/IR26:31;
   next: 18.
```

*A complete listing of the control sequence for a hard-wired controller implementation of MODEL is found in Appendix A. To follow the design from this section, ignore the parentheses around step numbers in the appendix.

For Page zero addressing, the address field ($IR_{16:31}$) is zero-extended (that is, leading 0's are added to make the number 32 bits). For relative addressing, the sign-extended address field is added to the program counter (which had already been incremented to point to the next instruction at step 2). Both produce an address and thus branch to step 18.

Page Zero

```
10. EA ← 0000, IR16:31;
    next: 18.
```

Relative

```
11. IR16: EA ← ADD1:32[FFFF, IR16:31; PC; 0]
    IR16′: EA ← ADD1:32[0000, IR16:31; PC; 0]
    next: 18.
```

The remaining three address modes all require a second word. At steps 13 and 14, the second word of the instruction is read into EA, and the program counter is incremented. For direct addressing, the second word is the effective address and is sent to EA. For indirect addressing, the second word is the address where the effective address will be found. Thus, after the second word is read at step 13, that memory location is read (step 15) and its contents are sent to EA.

Read Second Word, Direct

```
13. AD = PC; read = 1; EA ← DATA.
14. PC ← INC[PC];
    next/IR13:15: 18, 15, 1, 1, 16, 1, 1, 1.
```

Indirect

```
15. AD = EA; read = 1; EA ← DATA;
    next: 18.
```

For immediate addressing, the second word is the data and is sent to WORK, and control branches to step 20 (since a read is not needed for data).

Immediate

```
16. WORK ← EA;
    next: 20.
```

At step 18, we read the data from the effective address register, EA, into WORK. We could have a branch to skip this step for those instructions that do not require data (such as store, jump, and call).

Data Read

```
18. AD = EA; read = 1; WORK ← DATA;
    next: 20.
```

EXAMPLE 10.4

We want to add two new addressing modes; register indirect with auto-post-incrementing, and short immediate. Assume that step 4 branches to steps 7 and 9 for these. Register indirect with auto-post-incrementing produces the same address as register indirect, but also increments the register after using it. For short immediate, the address field is sign-extended.

Register Indirect with Auto-Post-Incrementing

```
7. EA ← REG/IR26:31.
8. REG/IR26:31 ← INC[REG/IR26:31];
   next: 18.
```

Short Immediate

```
9. IR16: WORK ← FFFF, IR16:31;
   IR16': WORK ← 0000, IR16:31;
   next: 20.
```

The details of the instruction decode step (or steps) are not shown, since we did not specify the coding of the op-code and we are implementing the controller for only a few instructions. We will show the implementation of the individual instructions, using step numbers beginning at 25.

Load register requires only one step, after which it returns to step 1 to fetch a new instruction. On *Store*, the result goes to the register specified by $IR_{26:31}$ for register addressing ($IR_{12}' \cdot IR_{13}' \cdot IR_{14}' \cdot IR_{15}'$) and to memory for all other types. Store does not permit immediate addressing; it is treated as a no-op.

LOD

```
25. REG/IR6:11 ← WORK; z ← (OR[CPUBUS]';
    n ← CPUBUS0;
    next: 1.
```

STO

```
26. WORK ← REG/IR6:11.
27. next: 28 (IR12' · IR13' · IR14' · IR15'), 1
    (IR12 · IR13 · IR14' · IR15'), 29 (else).
28. REG/IR26:31 ← WORK;
    next: 1.
29. ADIN = EA; DATA = WORK; write = 1;
    next: 1.
```

Push and *Pop* are implemented after a decoding branch from step 60. *Push* writes to the location pointed to by the stack pointer and then decrements the pointer. *Pop* first increments the stack pointer and then reads.

PSH

65. ADIN = SP; DATA = REG/$IR_{6:11}$; **write** = 1.
66. SP ← DEC[SP];
    next: 1.

POP

67. SP ← INC[SP].
68. ADIN = SP; **read** = 1; REG/$IR_{6:11}$ ← DATA;
    z ← (OR[CPUBUS])′; **n** ← $CPUBUS_0$;
    next: 1.

The addition and subtraction instructions are each only one step. The adder has a 33-bit output, the left bit being the carry. Two's complement overflow is detected when two numbers have the same sign (INA_0 = INB_0) and the result has the opposite sign ($CPUBUS_0$). The expressions for each of the flag bits is the same for almost all instructions.

ADD

30. **c,** REG/$IR_{6:11}$ ← ADD[REG/$IR_{6:11}$; WORK; 0];
    z ← (OR[CPUBUS])′; **n** ← $CPUBUS_0$;
    v ← $INA_0 \cdot INB_0 \cdot CPUBUS_0' + INA_0' \cdot INB_0' \cdot CPUBUS_0$;
    next: 1.

ADC

31. **c,** REG/$IR_{6:11}$ ← ADD[REG/$IR_{6:11}$; WORK; **c**];
    z ← (OR[CPUBUS])′; **n** ← $CPUBUS_0$;
    v ← $INA_0 \cdot INB_0 \cdot CPUBUS_0' + INA_0' \cdot INB_0' \cdot CPUBUS_0$;
    next: 1.

SUB

32. **c,** REG/$IR_{6:11}$ ← ADD[REG/$IR_{6:11}$; WORK′; 1];
    z ← (OR[CPUBUS])′; **n** ← $CPUBUS_0$;
    v ← $INA_0 \cdot INB_0 \cdot CPUBUS_0' + INA_0' \cdot INB_0' \cdot CPUBUS_0$;
    next: 1.

CMP

```
34. c ← ADD0[REG/IR6:11; WORK′; 1];
    CPUBUS = ADD1:32[REG/IR6:11; WORK′; 1];
    z ← (OR[CPUBUS])′; n ← CPUBUS0;
    v ← INA0 · INB0 · CPUBUS0′ + INA0′ · INB0′
     · CPUBUS0;
    next: 1.
```

The next two instructions operate on numbers in WORK and ignore the RN field. The results are stored back in either a register or memory, using the steps already implemented for STO.

INC

```
36. WORK ← ADD1:32[00000001; WORK;0];
    z ← (OR[CPUBUS])′; n ← CPUBUS0;
    next: 27.
```

NOT

```
40. WORK ← WORK′; z ← (OR[CPUBUS])′;
    next: 27.
```

AND uses operands from a register and WORK, storing the result back in that register.

AND

```
42. REG/IR6:11 ← REG/IR6:11 · WORK;
    z ← (OR[CPUBUS])′; n ← CPUBUS0;
    next: 1.
```

The instruction decode step reaches step 43 for all of the shifts and rotates. We assume that the op-code for ASR ends in 00, and ROR ends in 10. We shift one place at a time; step 43 transfers to the step for the store instruction when $IR_{7:11}$ counts down to 0. $IR_{7:11} = 00000$ is treated as zero, making these instructions no-ops.

```
43. IR7:11 ← DEC[IR7:11];
    next: 44 (OR[IR7:11]), 27 (else).
44. next: 45 (IR4), 47 (else).
```

ASR

```
45. WORK ← WORK0, WORK0:30;
    z ← (OR[CPUBUS])′;
    next: 43.
```

ROR

```
47. WORK ← WORK31, WORK0:30;
    next: 43.
```

If faster shifting were required, we could build a *barrel shifter*, which would allow a shift of any number of places in one step. The hardware for that is more complex, since each bit of WORK could be loaded with any other bit or with 0. In contrast, the implementation we have shown only requires that each bit be loaded with the bit on either side (or 0 for the first and last bits).

The conditional jump and call instructions depend on the variable **br,** where

$$\mathbf{br} = \mathrm{OR}[\mathrm{DCD}(\mathrm{IR}_{8:11}) \cdot (\mathbf{z, z', n, n', c, c', v, v', 0, 0, 0, 0, 0, 0, 0, 1})]$$

DCD is a decoder with four inputs and 16 outputs, one of which is 1. That is ANDed with the 16-bit vector with each of the conditions, as specified in Table 10.1. Thus, **br** is 1 if the specified branch condition is satisfied and 0, otherwise. (Unused codes are treated as never branch. They could be treated as an unconditional branch by changing all of the 0's to 1's.)

On jump, the program counter is loaded with the effective address if the condition is met; otherwise, it returns to step 1. On a successful call, the contents of the program counter are first pushed onto the stack and then the effective address is moved to the PC.

JMP

```
50. br: PC ← EA;
    next: 1.
```

CLL

```
51. next: 1 (br), 52 (else)
52. ADIN = SP; DATA = PC; write = 1.
53. SP ← DEC[SP].
54. PC ← EA;
    next: 1.
```

Finally, the return from subroutine (unconditional) pops the address from the stack and loads that into the program counter.

RTS

```
70. SP ← INC[SP].
71. ADIN = SP; read = 1; PC ← DATA;
    next: 1.
```

EXAMPLE 10.5

Add a new instruction to decrement the register pointed to by the RN field and jump to the address if the result is 0. (This is a loop control instruction.)

The simplest way is to do the addressing and jump to step 55 from step 20.

```
55. WORK ← REG/IR6:11.
56. REG/IR6:11 ← ADD1:32[FFFFFFFF; WORK; 0];
    next: 1 (OR[CPUBUS]), 57 (else).
57. PC ← EA;
    next: 1.
```

Since step 57 is identical to step 54, the branch at step 56 could go to 54, eliminating step 57. It was necessary to move the register to `WORK` (step 55) because the bus structure up until this point put both the register and the constant on `INA`. If the bus structure were modified, we could combine steps 55 and 56.

EXAMPLE 10.6

Add a new instruction, *Stack add*. It adds the top two entries on the stack and pushes the answer back onto the stack. The operands are destroyed. No flags are involved.

This requires and additional register, `WORK2`, to store the second number. (That register would be needed by more complex instructions, such as *Multiply*. We pop the two operands, add them, and then push the result onto the stack.

```
80. SP ← INC[SP].
81. ADIN = SP; read = 1; WORK ← DATA.
82. SP ← INC[SP].
83. ADIN = SP; read = 1; WORK2 ← DATA.
84. WORK ← ADD1:32[WORK2; WORK; 0].
85. ADIN = SP; DATA = WORK; write = 1.
86. SP ← INC[SP];
    next: 1.
```

EXAMPLE 10.7

Modify **br** so as to provide conditional branches for comparing two signed and unsigned numbers.

Two numbers can be compared by subtracting them and then using a conditional jump. The comparisons are between the number in the register ($REG/IR_{6:11}$), *a*, compared with the number specified by the address, *b*. There are separate tests for signed and unsigned numbers based on the flag bits.

For signed numbers, when $a < b$, the result is negative unless there is overflow. Thus, the condition is $\mathbf{n} \oplus \mathbf{v}$. For less than or equal, we have $\mathbf{z} + \mathbf{n} \oplus \mathbf{v}$. The opposite of less than is greater than or equal, $(\mathbf{n} \oplus \mathbf{v})'$, and greater than is the complement of less than or equal, $\mathbf{z} + \mathbf{n} \oplus \mathbf{v}$. For unsigned numbers, $a < b$ is indicated by $\mathbf{c}'$, less than or equal by $\mathbf{c}' + \mathrm{z}$, greater than by $(\mathbf{c}' + \mathbf{z})'$, and greater than or equal by $\mathbf{c}$.

If the following codes are added to the list in Table 10.1,

1000	$<$	(After SUB for signed numbers)
1001	$\leq$	
1010	$\geq$	
1011	$>$	
1100	$\leq$	(After SUB for unsigned numbers)
1101	$>$	

then the definition of *br* becomes

$$\mathbf{br} = \mathrm{OR}[\mathrm{DCD}(\mathrm{IR}_{8:11}) \cdot (\mathbf{z}, \mathbf{z}', \mathbf{n}, \mathbf{n}', \mathbf{c}, \mathbf{c}', \mathbf{v}, \mathbf{v}', \mathbf{n} \oplus \mathbf{v}, \mathbf{z} + \mathbf{n} \oplus \mathbf{v}, (\mathbf{n} \oplus \mathbf{v})', (\mathbf{z} + \mathbf{n} \oplus \mathbf{v})', \mathbf{c}' + \mathbf{z}, \mathbf{c} \cdot \mathbf{z}', 0, 1)]$$

EXAMPLE 10.8

Include a new instruction to add a set of numbers stored in consecutive memory locations. The address field specifies the location of the first number. The register specified by RN contains the size of the set (how many numbers to be added) and is replaced by the sum. The flag bit **c** is set to one iff any of the additions produced unsigned overflow.

Step 20 branches to step 90. We need three registers, in addition to EA, in this process: one to hold the count, one to hold the sum as we are adding, and one to hold each new number as it is read. One approach is to store the sum in the register and the count in WORK. The additional register, WORK2, would be connected to INB. (Note that this is a different connection from Example 10.6.)

```
90. WORK ← REG/IR6:11;
    next: 91 (OR[CPUBUS]), 1 (else).
91. REG/IR6:11 ← 00000000; c ← 0.
92. AD = EA; read = 1; WORK2 ← DATA.
93. REG/IR6:11 ← ADD1:32[REG/IR6:11; WORK2; 0];
    c ← c + ADD0[REG/IR6:11; WORK2; 0].
94. WORK ← DEC[Work];
    next: 95 (OR[CPUBUS]), 1 (else).
95. EA ← INC[EA];
    next: 92.
```

On the first step, if the count is 0, the register already has the sum of 0 and the instruction is complete. When the count goes to 0, the process is complete. Since the sum is already in the register, we can go back to step 1 to fetch a new instruction. Note that EA must be connected to INA to implement step 95. (That connection was not previously required.)

[SP 5, 6, 7, 8; EX 4, 5, 6, 7, 8, 9]

10.3 IMPLEMENTATION OF MODEL CONTROL SEQUENCE WITH A HARDWIRED CONTROLLER

The simplest implementation is to use a one-hot controller (where each step corresponds to one flip flop). One flip flop of the controller has a 1 in it, and all others have a 0 (similar to the controller of Figure 9.9).

Such a controller for the instruction fetch and addressing portion of the control sequence is shown in Figure 10.4. So as to make the figure readable, the clock input line to each flip flop has been omitted, as have the output signal lines from each of the flip flops (There are no outputs from steps 3 and 4, which are only branches). Note at steps 11, there is a pair of AND gates for the conditional data transfers.

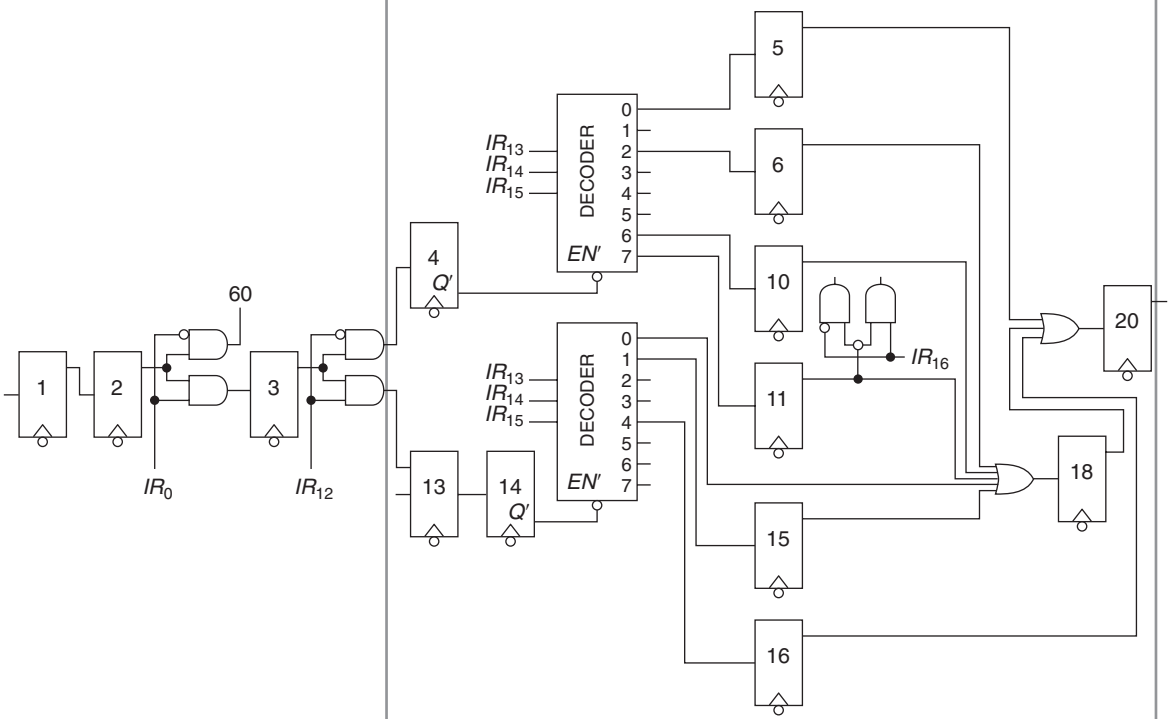
The eight-way decoder for the 1-word instructions is enabled by the Q' output of flip flop 4. One of the outputs of that decoder is active when the controller is in step 4, putting a 1 into one of flip flops 5, 6, 10, or 11 at the next clock. (The unused addressing code outputs (1, 3, 4, and 5) are not shown connected, but would all go to an OR gate at the input of flip flop 1.) A second eight-way decoder for the 2-word instructions is enabled by the Q' output of flip flop 14.

From the controller block diagram, it is easy to see that register addressing takes five clocks to reach step 20 (steps 1, 2, 3, 4, and 5). (For the purpose of timing discussions, we will include the time in step 20 in the execution portion.) Register indirect (step 6), Page zero (step 10), and Relative (step 11) each require five clocks (steps 1, 2, 3, 4, and one of 6, 10, or 11) to reach step 18 and a sixth to reach step 20. Direct addressing takes six clocks (steps 1, 2, 3, 13, 14, and 18). Indirect uses a seventh clock period (step 15). Immediate takes six clocks. (It uses step 16, but does not need step 18.)

The speed of the system can be greatly increased by taking advantage of undelayed steps. Since IR was not changed at steps 3 and 4, step 3 can be undelayed. Next, we note that either step 4 or all of the steps reached directly from step 4 can be made undelayed. That would mean that steps 5, 6, 10, and 11 would be executed at the same time as the branch at step 4, which does not change any registers nor utilize the internal bus. Steps 1 to 18 become

```
1. AD = PC; read = 1; IR ← DATA.
2. PC ← INC[PC];
```

Figure 10.4 Controller for instruction fetch and addressing.




```
      next: 60 (IR0), 3. (else).
(3.)  next: 12 (IR12), 4 (else)
4.    next/IR13:15: 5, 1, 6, 1, 1, 1, 10, 11.
(5.)  WORK ← REG/IR26:31;
      next: 20.
(6.)  EA ← REG/IR26:31;
      next: 18.
(10.) EA ← 0000, IR16:31;
      next: 18.
(11.) IR16: EA ← ADD1:32[FFFF, IR16:31; PC; 0]
      IR16′: EA ← ADD1:32[0000, IR16:31; PC; 0]
      next: 18.
13.   AD = PC; read = 1; EA ← DATA.
14.   PC ← INC[PC];
      next/IR13:15: 18, 15, 1, 1, 16, 1, 1, 1.
```

```
15.   AD = EA; read = 1; EA ← DATA;
      next: 18.
16.   WORK ← EA;
      next: 20.
18.   AD = EA; read = 1; WORK ← DATA;
      next: 20.
```

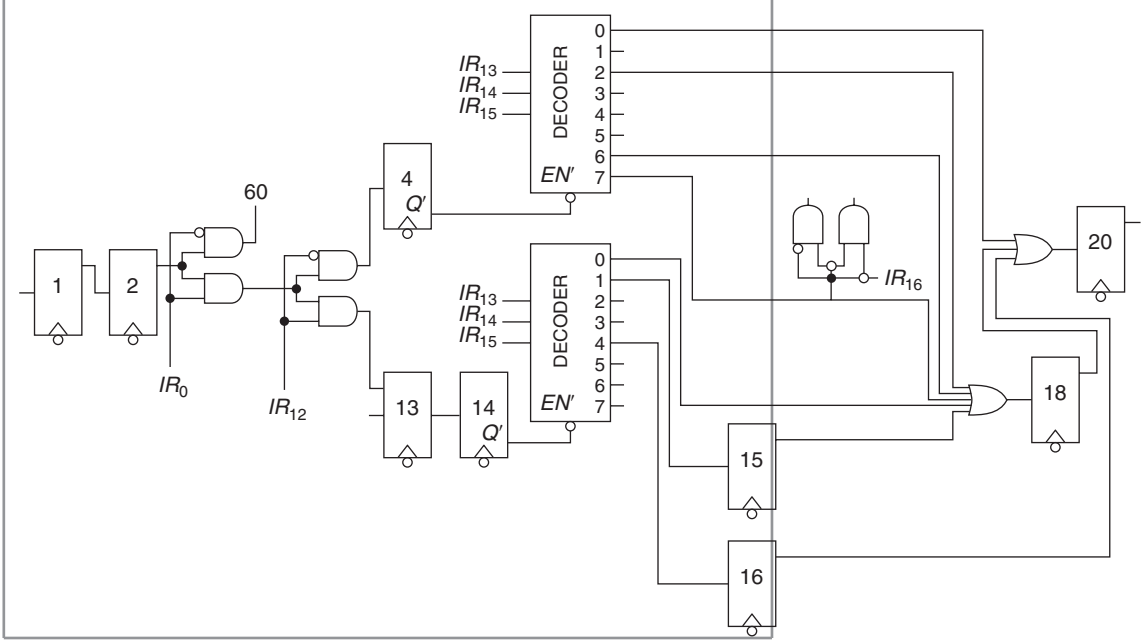
This reduces the execution time for all 1-word addressing modes by two clocks and the 2-word modes by one clock. That eliminates five (of the 13) flip flops associated with steps 1 to 18. The controller is shown in Figure 10.5.

One other small timing improvement could be made by checking at step 18 whether it was necessary to fetch data. In particular, *Jump*, *Call*, and *Store* do not require data. If the op-codes for those instructions (and only those instructions) have a 1 in bit 2, then we could rewrite

```
(18.) next: 20 (IR2), 19 (else).
19.   AD = EA; read = 1; WORK ← DATA.
```

That would save one clock time for these instructions.

Figure 10.5 Controller for instruction fetch and addressing.



We could make step 20 undelayed. But then the first step of the execution part of any of the instructions would be delayed, because they all involve using the bus (which was also used in each of the steps leading to step 20). If step 20 is delayed, we can save a number of flip flops in the controller by making the first step of each memory reference instruction (24, 26, 30, . . .) undelayed.* Most instructions would take only one clock time for execution (that at step 20). Exceptions include those that require a store (STO, NEG, INC, DEC, and NOT), which have a second clock at either step 28 or 29, and CLL, which requires two extra clock times to push the program counter onto the stack. The shifts and rotates also require two clocks for each place plus one extra as it leaves the loop at step 43.† The DDL for MODEL is shown in Appendix A.

The timing of the instructions is summarized in Table 10.2. The dashes indicate that this addressing mode is not allowed for those instructions.

Table 10.2 Timing of Instructions.

Instruction	(REG),PG-0				
	REG	**REL**	**DIR**	**IND**	**IMM**
LOD,ADD,ADC, SUB,CMP,AND	4	5	6	5	5
STO,INC,NOT	5	6	7	6	—
JMP	—	5	6	5	—
CLL	—	7	8	7	—
ASR,ROR	$5 + 2n$	$6 + 2n$	$7 + 2n$	$6 + 2n$	—
PSH,POP,RTS	3 (no addressing)				

*This approach results in the machine being slower by one clock for failed conditional jumps and calls. We could rewrite step 50 as

```
(50.)   next: 50a (br), 1 (else).

50a.    PC ← EA;

        next: 1.
```

Thus, if step 20 were undelayed, step 50 could still be undelayed, and failed conditional jumps (and calls if we also modified step 51 in the same way) would take no clock times for execution.

†If we build a separate decrementer to count, we could make step 43 undelayed and would only need one clock time per place shifted.

EXAMPLE 10.9

We will look at the timing for the new addressing mode and instructions described in Examples 10.4, 10.5, 10.6, and 10.8.

10.4: Steps 7 and 9 can be undelayed. For the controller of Figure 10.5, Register Indirect with Auto-post-increment would take one more clock time than register indirect. Short immediate would take the same time as Register addressing.

10.5: Steps 55 can be undelayed. From step 20, this will take two clock times if it does not jump (Steps 20 and 56) or three clock times if it does jump (Steps 20, 56, and 57).

10.6: Only step 80 can be undelayed. This will take eight clock times (including steps 1 and 2).

10.8: Step 90 can be undelayed, requiring one clock time if there are zero numbers to add. Step 91 is executed once, and steps 92, 93, and 94 are each executed n times and step 95 is executed $n - 1$ times, for a total of $4n + 1$ (in addition to the time for steps 1 to 18).

EXAMPLE 10.10

In order to speed the machine, one thought was to add an incrementer for PC that does not use the bus. (Thus, instead of PC receiving all its inputs from CPUBUS, there would be a multiplexer on the input to PC.)

Steps 4 and 14 could then be made undelayed (or 4, 15, and 16), since the only reason that they are delayed is that the incrementing of PC used the bus. Step 2 cannot be made undelayed, because the branch uses the data being loaded into IR at step 1.

This would reduce the execution time of all instructions by one clock.

[SP 9, 10, 11; Ex 10, 11, 12, 13]

10.4 MODEL WITH A SLOWER MEMORY

If the main memory always took a fixed number of clock times, we could modify the read and write steps accordingly. Say that we could connect the address to ADIN and put a 1 on **read** at one clock time, and the contents of that location would be on DATA two clock times later. Then, steps 1, 2, 3, and 4 might be rewritten

```
1.      AD = PC; read = 1.
2.      PC ← INC[PC];
3.      IR ← DATA.
4.      next: 60 (IR0), 5. (else).
(4a.)   next: 12 (IR12), 6 (else)
(4b.)   next/IR13:15: 5, 1, 6, 1, 1, 1, 10, 11.
```

Of course, the remaining steps would need to be renumbered. This only adds one clock time, since now PC can be incremented during the read process.

Steps 13 and 14 would now become

```
13.  AD = PC; read = 1.
14.  PC ← INC[PC].
14a. EA ← DATA;
     next/IR13:15: 18, 15, 1, 1, 16, 1, 1, 1.
```

again, adding one clock time.

The write at step 29 would become

```
29.  ADIN = EA; DATA = WORK; write = 1.
29a. .
29b. next: 1.
```

The other steps involving memory (for *Call*, *Return*, *Push*, and *Pop*) would also have to be rewritten.

If memory required that the address and **read** be kept 1 for all three clock times, that would not change the timing. Steps 1 to 3 would become

```
1.   AD = PC; read = 1.
2.   AD = PC; read = 1; PC ← INC[PC];
3.   AD = PC; read = 1; IR ← DATA.
```

If write also required the signals to remain, then step 29 would become

```
29.  ADIN = EA; DATA = WORK; write = 1.
29a. ADIN = EA; DATA = WORK; write = 1.
29b. ADIN = EA; DATA = WORK; write = 1;
     next: 1.
```

If the amount of time for a read and write of memory were variable (possibly depending on memory being used by another component), there would need to be a signal from memory, such a **memready.** If we must hold the inputs to memory until there is an answer, then step 1 is replaced by

```
1.   AD = PC; read = 1;
     next: 2 (memready), 1 (else).
(2.) IR ← DATA.
```

PC cannot be loaded until the next clock time. If the memory inputs were only required during the first clock period, we could replace steps 1 by

```
1.      AD = PC; read = 1;
(2.)    next: 4 (memready), 3 (else).
3.      next: 2.
4.      IR ← DATA.
```

For these two cases, the write at step 29 would become

```
29.     ADIN = EA; DATA = WORK; write = 1;
        next: 1 (memready), 29 (else).
```

or

```
29.     ADIN = EA; DATA = WORK; write = 1.
(29a.)  next: 1 (memready), 29b (else).
29b.    next: 29a.
```

10.5 A MICROPROGRAMMED CONTROLLER

In this section, we will examine the ideas behind the design of microprogrammed controllers. Such a controller replaces the sequential circuit of a hard-wired controller by a small amount of control logic and a special memory in which a representation of the control sequence (the DDL) is stored. The steps of that sequence (the microinstructions) are fetched and executed by the control logic.

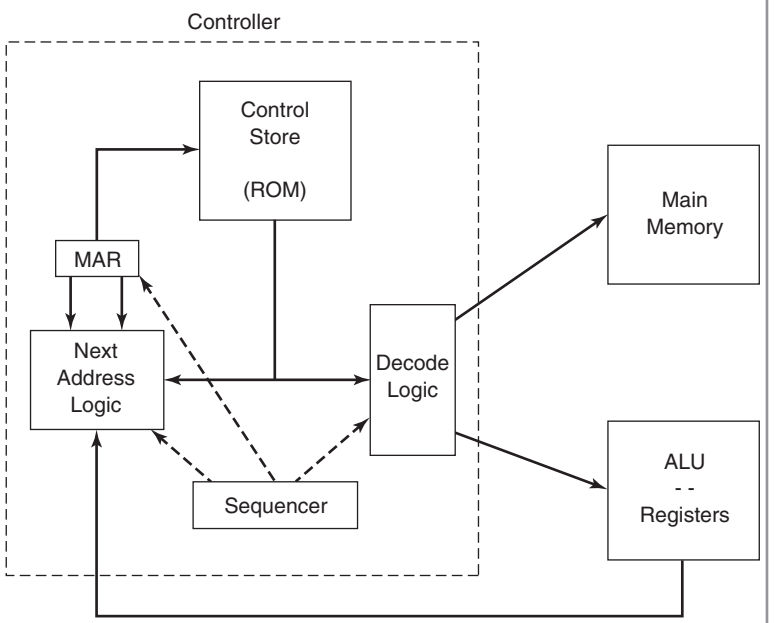
> ASIDE: A computer with a microprogrammed controller has two distinct memories: the main memory of the computer in which instruction and data are stored, and the special memory, often referred to as the control store, where the representation of the control sequence is stored. These memories are independent and are typically different sizes. The control store is often a Read-Only Memory (ROM), since the control sequence that is stored in it rarely, if ever, changes.

The block diagram of Figure 10.6 shows the basic structure of a microprogrammed controller and its connections to the rest of the machine. Those connections are identical for both the hard-wired and microprogrammed controller. It is only the structure of the inside of the controller box that has changed.

The MAR contains the location in the control store of the current microinstruction (the one that is being executed during this clock period). The output of the ROM is a set of lines containing that microinstruction. As each instruction is executed, the *next address logic* produces the address of the next microinstruction (DDL step). Often, that logic just performs incrementation. For data transfers and connections, the *decode logic* produces the appropriate control signals for the rest of the machine.

The sequencer is a very small hard-wired controller that controls the fetching and sequencing of the microinstructions from the control store.

Figure 10.6 A microprogrammed controller.



It contains only one (or at most two or three) flip flops. The decode logic, which produces the signals to the rest of the machine, depends only on what data transfers and connections are required. The next address logic is based on the branch conditions in the DDL steps. Both logic blocks are fairly simple. Neither depends on the details of the control sequence. That sequence is stored (in coded form) in the ROM. One advantage of microprogramming is that the controller logic is simpler than for the hard-wired version. The largest part of the controller is the control store. But that can be implemented using a rather inexpensive off-the-shelf ROM. The hard-wired controller for a large computer consists of an irregular collection of flip flops and gates, which must be fabricated from scratch for each new controller.

Another advantage of microprogramming becomes apparent when the designer wishes to make a modification in the control sequence. This may occur because of an error in the original version or because a new or modified instruction is being introduced. We must then rewrite a portion of the DDL sequence. If this has been implemented in a hard-wired controller, then the logic must be changed and a new sequencer fabricated. If the sequencer was implemented on a VLSI chip, the old chip is now useless and a new one must be produced. On the other hand, in a microprogrammed controller, the modifications are usually easier. If, as is usually the case, the data movements and branch conditions required for this change were already implemented (either because they were

needed for some other instruction or because the hardware designer provided extra features for possible later expansion), then no hardware modifications need be made. All that must be done is that a new sequence must be loaded into the control store. ROM programmers allow that to be done quickly and inexpensively.

The major disadvantage of microprogramming is speed. A microprogrammed machine is usually slower than a hard-wired one. This is partly a result of the limitations of how much can be stored in a single word in the control store. Whereas a DDL step can contain an unlimited number of data transfers and connections, as well as a multiway branch and conditional data transfers, it is not practical to allow for all of these possibilities in the coding of microinstructions. Thus, some DDL steps will result in two or more steps in a microprogrammed implementation. A second factor is that each step must be obtained from memory; that takes a clock time. Those steps that are undelayed in a hard-wired implementation do require a clock time in a microprogrammed machine.

10.6 SOLVED PROBLEMS

1. We have two new address types in MODEL

a. Page zero indirect

b. Indexed, where $IR_{26:31}$ specify which register is used as an index register. The contents of the second word is added to the index register.

Specify what happens on an instruction that loads REG^5 using the following values:

This instruction is at location 10000000

The second word of the instruction (if any) is 12345678

Bits 16 to 31 of this instruction word are 9ABC

REG^4 = FFEE0000

REG^{3C} = 22446688

M[00009ABC] = 12345678

M[12345678] = FDECBA98

M[3478BD00] = 11223344

a. $REG^5 \leftarrow$ FDECBA98 PC $\leftarrow$ 12341235

The Page zero address (00009ABC) contains the address of the data to be loaded.

b. Address = 22446688 + 12345678 = 3478BD00

$REG^5 \leftarrow$ 11223344 PC $\leftarrow$ 12341236

The index register is specified by the last 6 bits of the first instruction word ($11\ 1100_2 = 3C_{16}$)

2. We have an instruction to store REG^4 at the location specified by the address field. Assume the values of Solved Problem 1. What registers and memory locations are changed for each of the address types?

a. Register
b. Register indirect
c. Page zero
d. Relative
e. Direct
f. Indirect
g. Immediate

a. Register $REG^{3C} \leftarrow$ FFEE0000 PC $\leftarrow$ 10000001
b. Register indirect M[22446688] $\leftarrow$ FFEE0000
PC $\leftarrow$ 10000001
c. Page zero M[00009ABC] $\leftarrow$ FFEE0000 PC $\leftarrow$ 10000001
d. Relative EA = 10000000 + 1 + FFFF9ABC = 0FFF9ABD
M[0FFF9ABD] $\leftarrow$ FFEE0000 PC $\leftarrow$ 10000001
e. Direct M[12345678] $\leftarrow$ FFEE0000 PC $\leftarrow$ 10000002
f. Indirect M[FDECBA98] $\leftarrow$ FFEE0000 PC $\leftarrow$ 10000002
g. Immediate PC $\leftarrow$ 10000001 (This is treated as a no-op since Immediate is not allowed for stores.)

3. For each of the following instructions, what registers, flag bits, and memory locations are changed for each of the address types? Assume the following initial values for each instruction:

This instruction is at location 10000000
$z = 0$ $n = 1$ $c = 1$ $v = 0$
REG^5 = 12345678
REG^{12} = 22446688
REG^{63} = 88888888
M[00009ABC] = EDCBA988
M[12345678] = 2468ACE0
M[20001000] = 7E110000
M[7E110000] = 345678FF
M[88888887] = 00112233
M[88888888] = 11223344
M[88888889] = 22334455

a. STO REG^{12}, (REG^5)
b. PSH REG^5

c. POP REG^5
d. ADD REG^5, 20001000
e. ADC REG^5, 20001000
f. ADD REG^5, z9ABC
g. SUB REG^5, REG^{63}
h. CMP REG^{12}, 88888888
i. INC 00009ABC
j. NOT REG^{12}
k. AND REG^5, (20001000)
l. ASR 3, REG^{63}
m. ASR 5, 20001000
n. ROR 8, (REG^5)
o. JMP (REG^5)
p. JM4 (REG^5)
q. JM5 12341234
r. CLL (REG^5)
s. RTS

a. Register indirect, EA = 12345678
M[12345678] ← 22446688 PC ← 10000001
b. M[88888888] ← 12345678 REG^{63} ← 88888887
PC ← 10000001
c. REG^{63} ← 88888889 REG^5 ← 22334455 $z \leftarrow 0$ $n \leftarrow 0$
PC ← 10000001
d. 12345678
7E110000
90455678
REG^5 ← 90455678 $z \leftarrow 0$ $n \leftarrow 1$ $c \leftarrow 0$ $v \leftarrow 1$
PC ← 10000002

Note that there is signed number overflow since both operands begin with a 0 (binary) but the result begins with a 1.

e. REG^5 ← 90455679 $z \leftarrow 0$ $n \leftarrow 1$ $c \leftarrow 0$ $v \leftarrow 1$
PC ← 10000002
f. 12345678
EDCBA988
(1) 00000000
REG^5 ← 1234F134 $z \leftarrow 1$ $n \leftarrow 0$ $c \leftarrow 1$ $v \leftarrow 0$
PC ← 10000001

g. 1
12345678
77777777
89ABCDF0
REG[5] ← 89ABCDF0 $z \leftarrow 0$ $n \leftarrow 1$ $c \leftarrow 0$ $v \leftarrow 1$
PC ← 10000001

h. 1
22446688
EEDDCCBB (Bit-by-bit complement of 11223344)
(1) 11223344
$z \leftarrow 0$ $n \leftarrow 0$ $c \leftarrow 1$ $v \leftarrow 0$ PC ← 10000002

REG[12] is not changed.

i. M[00009ABC] ← EDCBA989 PC ← 10000002 $z \leftarrow 0$ $n \leftarrow 1$

Note that this uses direct addressing and requires a 2-word instruction. We could achieve the same result using Page zero addressing as in part (k).

j. REG[12] ← DDBB9977 $z \leftarrow 0$ PC ← 10000001

k. 12345678 AND 345678FF
REG[5] ← 10145078 $z \leftarrow 0$ $n \leftarrow 0$ PC ← 10000002

l. REG[63] ← 11111111 $z \leftarrow 0$ PC ← 10000001

m. M[20001000] ← 03F08800 $z \leftarrow 0$ PC ← 10000002

n. M[12345678] ← E02468AC $z \leftarrow 0$ PC ← 10000001

o. PC ← 12345678

p. PC ← 12345678 (since $c = 1$)

q. PC ← 10000002 (since $c = 0$)

It does not jump but likely reads the second word.

r. PC ← 12345678 M[88888888] ← 10000001
REG[63] ← 88888887

s. REG[63] ← 88888889 PC ← 22334455

4. For each part, a set of consecutive instructions are executed. Use the initial values of Solved Problem 3. Indicate what changes are made at the end of each set.

a. ADD REG[12], z9ABC
ADC REG[5], 7E110000

b. PSH REG[5]
PSH REG[12]
POP REG[1]

c. CLL 2000000
RTS

a. First instruction

22446688
EDCBA988

(1) 10101010 → REG^{12} $z \leftarrow 0$ $n \leftarrow 0$ $c \leftarrow 1$ $v \leftarrow 0$

Second instruction

1
12345678
345678FF
468ACF78

$REG^{12} \leftarrow 10101010$ $REG^{5} \leftarrow 468ACF78$ $PC \leftarrow 10000003$
$z \leftarrow 0$ $n \leftarrow 0$ $c \leftarrow 0$ $v \leftarrow 0$

b. First instruction

$M[88888888] \leftarrow 12345678$ $REG^{63} \leftarrow 88888887$

Second instruction

$M[88888887] \leftarrow 22446688$ $REG^{63} \leftarrow 88888886$

Third instruction

$REG^{63} \leftarrow 88888887$ $REG^{1} \leftarrow 22446688$ $z \leftarrow 0$ $n \leftarrow 0$

Note that M[88888887] still has 22446688.

c. First instruction

$PC \leftarrow 20000000$ $M[88888888] \leftarrow 10000002$
$REG^{63} \leftarrow 88888887$

Second instruction

$PC \leftarrow 10000002$ $REG^{63} \leftarrow 88888888$

Note that M[88888888] still has 10000002.

5. In Solved Problem 1, we discussed two additional address types. Show the changes in the DDL needed to implement these if Page zero indirect is coded 0101 and Indexed is coded 1010.

```
4. next/IR13:15: 5, 1, 6, 1, 1, 7, 10, 11.
```

Page Zero Indirect

```
7. EA ← 0000, IR16:31.
8. AD = EA; read = 1; EA ← DATA;
   next: 18.
```

Indexed

```
14.  PC ← INC[PC];
     next/IR13:15: 18, 15, 17, 1, 16, 1,
     1, 1.
17.  EA ← ADD1:32[REG/IR26:31; EA; 0].
```

This assumes that the EA register is connected to INA, which was not the case in Figure 10.3. Otherwise, we would need to move EA to WORK first.

```
17.  WORK ← EA
17a. EA ← ADD1:32[REG/IR26:31; WORK; 0];
     next: 18.
```

6. Another possible addressing mode is indirect with auto-pre-decrementing. Assume the branch at step 14 goes to step 17.

```
17.  WORK ← DEC[EA].
17a. AD = EA; DATA = WORK; write = 1.
17b. EA ← WORK;
     next: 18.
```

Note that we must preserve the second word of the instruction to store the decremented address there. Thus, the effective address is not loaded into EA until the last step.

7. We wish to add a new addressing type—multilevel indirect addressing. When an indirect address is fetched, the first bit is an indicator of whether that is the effective address or is still indirect. In this mode, all indirect addresses begin with the same bit as the address in the instruction. The branch at step 14 will branch to step 90 for this address type.

```
90.  AD = EA; read = 1; EA1:31 ← DATA1:31;
     next: 91 (DATA0), 18 (else).
```

Step 90 will be executed repeatedly until the first bit of the address being read is 0. This could result in an endless loop. To prevent that, some machines count how many times it loops, and terminate this after a fixed number of levels of indirection.

```
90.  WORK ← 00000000.
91.  AD = EA; read = 1; EA1:31 ← DATA1:31;
     next: 92 (DATA0), 18 (else).
92.  WORK ← INC[WORK];
```

```
        next: 1 (AND[CPUBUS28:31]), 91
        (else).
```

On the 15th pass through the loop, WORK contains 0000000E, and the right 4 bits of the incrementer output are all 1's, terminating the instruction (treating it as a no-op).

8. Design an instruction to multiply two unsigned numbers and store the result (or the 32 less significant bits) in the register from which the first operand comes. We will need to add two registers to implement this. The instruction sets the **c** flag if the answer does not fit into 1 word, and the **z** flag.

We initialize by setting a new 5-bit register, COUNT, to 0, putting one operand (the multiplier) in WORK, and putting the other (the multiplicand) in the new 32-bit register, WORK2. The partial product is stored in the register from which one operand came, which is also initialized to 0.

```
100.  WORK2 ← REG/IR6:11.
101.  REG/IR6:11 ← 00000000; COUNT ← 00;
      c ← 0.
102.  WORK ← 0, WORK0:30;
      next: 104 (WORK31'), 103 (else).
103.  REG/IR6:11 ← ADD1:32[REG/IR6:11; WORK2;
      0];
      z ← (OR[CPUBUS])';c ← c +
      ADD0[REG/IR6:11; WORK2; 0].
104.  COUNT ← INC[COUNT];
      next: 105 (OR[COUNT]), 1 (else).
105.  WORK2 ← WORK21:31, 0;
      next: 102.
```

If the right bit of the multiplier is 1, we add the multiplicand to the partial product, shifting the multiplier one place to the right. (Each time we come back to step 103, we will look at another bit of the multiplier.) COUNT keeps track of the number of bits (32). After adding, the multiplicand is shifted to the left (as we do in multiplication by hand). If there is a carry out of the adder at any stage, the c bit is set and remains set. Note that if there is overflow, we only get the lower 32 bits of the answer.

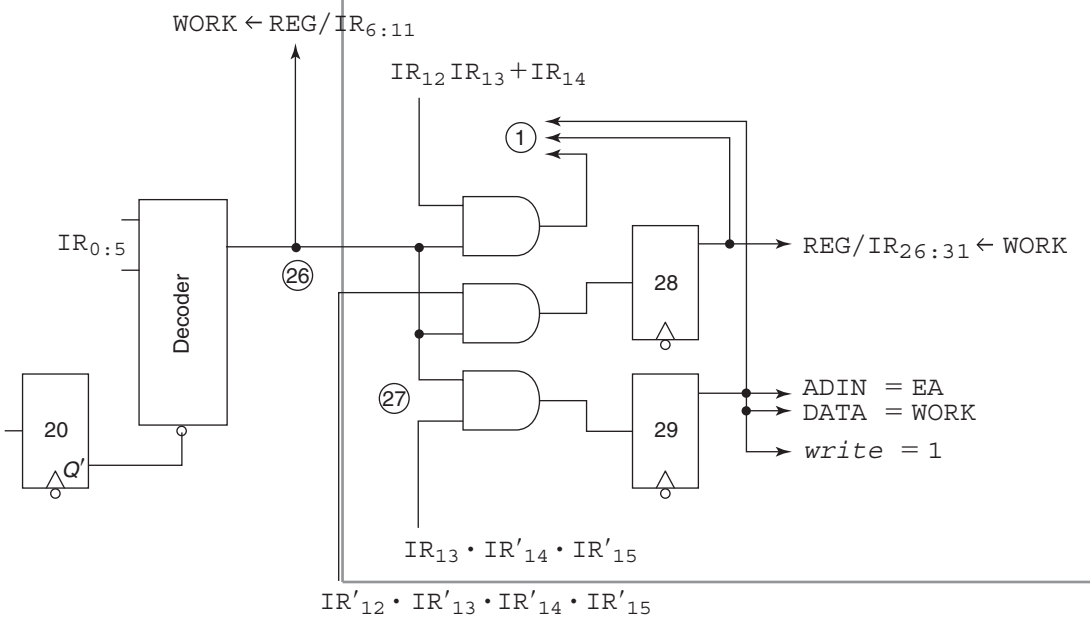
9. We wish to design a new instruction to perform double-precision addition. The first operand and the result come from a register pair, specified by the first 5 bits of the RN field. The low-order half of the number is stored in the odd register (register number ending in 1), and the high-order half is in the even register. The

other operand comes from two consecutive memory locations, where the low-order part comes from the location specified by EA and the high-order part comes from the previous location. (Register and Immediate addressing is not allowed.) Step 20 branches to step 100 for this instruction.

```
(100.) c, REG/(IR6:10, 1) ←
       ADD[REG/(IR6:10, 1); WORK; 0];
       z ← (OR[CPUBUS])'.
101.   EA ← DEC[EA].
102.   AD = EA; read = 1; WORK ← DATA.
103.   c, REG/(IR6:10, 0) ← ADD[REG/(IR6:10,
       c); WORK; 0];
       z ← z · (OR[CPUBUS])'; n ← CPUBUS0;
       v ← INA0 · INB0 · CPUBUS0' + INA0' ·
       INB0' · CPUBUS0;
       next: 1.
```

The carry from the first addition is used to make the second one an add with carry. The **z** flag is set only if both halves of the answer is 0. The **n** and **v** flags are determined by the most significant bit.

10. Using the controller design from Appendix A, show a block diagram of the hard-wired controller to implement the STO instruction.



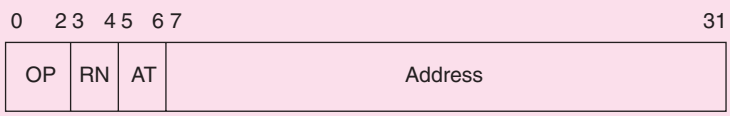
The flip flop and the decoder of step 20 are shown first. Steps 26 and 27 are undelayed, and thus there is no flip flop there. On completion of this instruction, the controller returns to step 1.

11. Compare the speed of double-precision addition, assuming direct addressing, for an ADD instruction followed by ADC with the double-precision instruction designed in Solved Problem 9.

ADD: 6 ADC: 6 Total: 12 (from Table 10.2)

In the new instruction, step 100 is undelayed, and thus it would take 9 clocks (1, 2, 13, 14, 18, 20, 101, 102, 103). If we had a special incrementer for EA (that does not use the bus), step 101 could also be undelayed, reducing the time to 8.

12. Design a small computer with the following instruction format:



where memory has 2^{25} words, 32 bits each. All instructions fit in a single word. There are four 32-bit registers (similar to the 64 in MODEL). The bus structure and the memory signaling is the same as in MODEL. There are no flag bits.

There are four addressing types

00 Direct
01 Immediate (sign-extended)
10 Relative to this instruction
11 Indirect (up to three levels)

Bit 0 of the address word indicates direct (0) or indirect (1). After three levels, bit 0 is ignored.

The operations are

000 Load
001 Increment (RN ignored)
010 Add
011 Subtract
100 Jump unconditional (RN ignored)
101 Jump conditional—only allows relative addressing

Based on number in register with condition specified by AT

00 = 0
01 > 0
10 ≠ 0

11 < 0
110 Store
111 not used in this problem

Write a complete DDL description of a hard-wired controller for this machine. Do not worry about unused or illegal combinations. Annotate your DDL. Make it run as fast as possible by making steps undelayed.

The first step here is

```
1. ADIN = PC; read = 1; IR ← DATA.
```

There is one adder for all addition, subtraction, and incrementing, as in MODEL. It has two 32-bit inputs and a carry-in. There is only a 32-bit output (no need for the carry-out).

Show a table indicating the execution time for each instruction and each addressing type.

```
SYSTEM NAME: NEW COMPUTER
     FLIP FLOPS: PC[0:24], IR[0:31],
     W[0:31], R[0:3; 0:31].
     COMMUNICATION BUSES: DATA[0:31],
     ADIN[0:24].
     INTERNAL BUSES: CPUBUS[0:31],
     INA[0:31], INB[0:31].
     OUTPUT LINES: read, write.
     1.  ADIN = PC; read = 1; IR ← DATA.
     2.  next: 5 (IR0 · IR1' · IR2), 3 (else).
     (3.)next/IR5:6: 10, 4, 5, 6.
```

Immediate

```
     (4.)IR7': W ← 00, IR7:31;
         IR7: W ← 7F, IR7:31;
         next: 12.
```

Relative

```
     (5.)IR7:31 ← ADD7:31[IR; 0^7, PC; 0];
         next: 10.
```

Indirect

(6.) ADIN = $IR_{7:31}$; **read** = 1; $IR_{7:31}$ ← $DATA_{7:31}$;
next: 7 ($CPUBUS_0$), 10 (else).

7. ADIN = $IR_{7:31}$; **read** = 1; $IR_{7:31}$ ← $DATA_{7:31}$;
next: 8 ($CPUBUS_0$), 10 (else).

8. ADIN = $IR_{7:31}$; **read** = 1; $IR_{7:31}$ ← $DATA_{7:31}$;
next: 10.

Data Fetch?

(10.) next: 12 (IR_0), 11 (else).

11. ADIN = $IR_{7:31}$; **read** = 1;
W ← DATA.

Decode

12. PC ← ADD[1^{32}; 0^7, PC; 0];
next/$IR_{0:2}$: 15, 16, 18, 19, 20, 21, 26, 1.

Load

15. R/$IR_{3:4}$ ← W;
next: 1.

Increment

16. W ← ADD[1^{32}; W; 0]

17. ADIN = $IR_{7:31}$; DATA = W; **write** = 1;
next: 1.

Add

18. R/$IR_{3:4}$ ← ADD[R/$IR_{3:4}$; W; 0];
next: 1.

Subtract

```
19. R/IR3:4 ← ADD[R/IR3:4; W'; 1];
    next: 1.
```

Jump

```
20. PC ← IR7:31;
    next: 1.
```

Jump Conditional

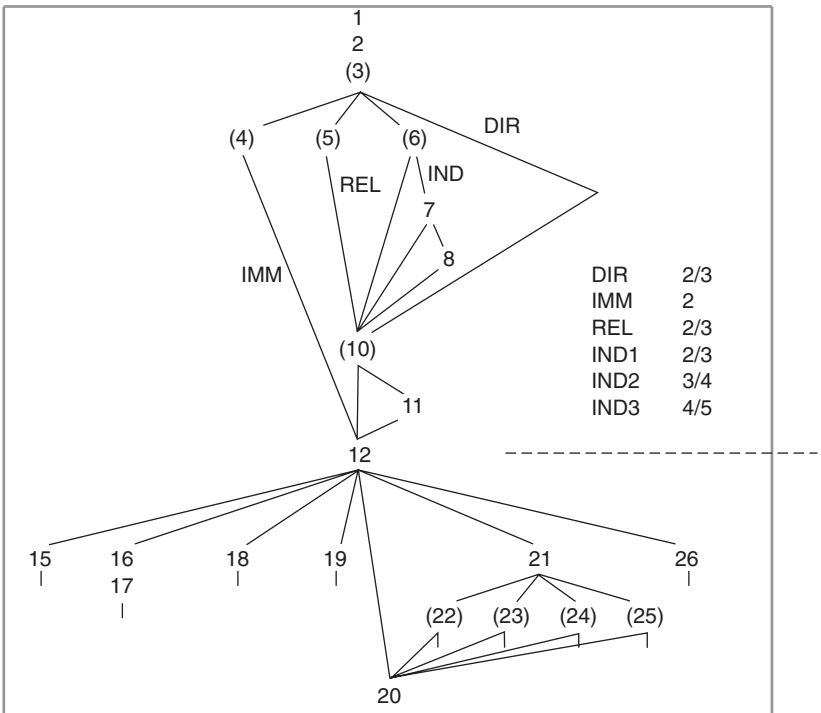
```
21. CPUBUS = R/IR3:4;
    next/IR5:6: 22, 23, 24, 25.
(22.) next: 1 (OR[CPUBUS]), 20 (else).
(23.) next: 20 (CPUBUS0' ·
      OR[CPUBUS]), 1 (else).
(24.) next: 20 (OR[CPUBUS]), 1 (else).
(25.) next: 20 (CPUBUS0), 1 (else).
```

Store

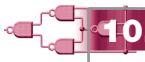
```
26. ADIN = IR7:31; DATA = R/IR3:4;
    write = 1;
    next: 1.

END DESCRIPTION
```

Note that we did not increment the Program Counter until after the addressing, since relative addressing is based on the address of the current instruction (not the next, as in MODEL).

The following graph shows the sequence of steps. The first step of the execution phase cannot be undelayed, since the decode step uses the bus while incrementing the PC.



DIR	5	6	4	—
IMMED	4	—	—	—
REL	5	6	4	5
IND 1	5	6	4	—
IND 2	6	7	5	—
IND 3	7	8	6	—



10.7 EXERCISES

1. For the following values (in MODEL):

This instruction is at location 76543210
The second word of the instruction (if any) is 22223333
Bits 16 to 31 of this instruction word are 4547
REG^1 = 79864322
REG^2 = 12341234
REG^7 = FFFFFF00
M[00000001] = 22222222
M[00004547] = 23423423
M[22223333] = 00000001
M[76547758] = 11223300
M[FFFFFF00] = 10101010

Show what registers are changed for each of the addressing types and each of these instructions (including the PC, but not the flag bits):

i. Load REG^1

*ii. Add to REG^2

a. Register
b. Register Indirect
c. Page zero
d. Relative
e. Direct
f. Indirect
g. Immediate

2. For each of the following instructions, what registers, flag bits, and memory locations are changed for each of the address types? Assume the following initial values for each instruction:

PC = 11111111
$z = 1 \quad n = 1 \quad c = 1 \quad v = 1$
REG^1 = 12345678
REG^2 = FFFFFFFF
REG^3 = 87654321
REG^{3F} = FFFFFFF0
M[00001234] = 00000010
M[10000000] = 91110000

M[12345678] = 2468ACE0
M[2468ACE0] = 3456789A
M[FFFFFFEF] = 00112233
M[FFFFFFF0] = 11223344
M[FFFFFFF1] = 22334455

a. i. STO REG^2, (REG^3)
ii. STO REG^1, 10000000
iii. STO REG^3, z1000
iv. STO REG^2, REG^3
v. LOD REG^2, REG^3

b. i. PSH REG^1
ii. POP REG^1

c. i. ADD REG^1, REG^2
ii. ADD REG^2, REG^1
iii. ADD REG^1, #34565432
iv. ADD REG^1, 10000000
v. ADD REG^3, 10000000

d. i. ADC REG^2, (REG^{3F})
ii. ADC REG^2, z1234
iii. ADC REG^2, #FFFFFFFF

e. i. SUB REG^2, 00001234
ii. SUB REG^1, REG^2

f. i. CMP REG^3, 2468ACE0
ii. CMP REG^3, #87654321
iii. CMP REG^3, REG^2

g. i. INC 10000000
ii. INC REG^2
iii. INC (REG^1)

h. i. NOT REG^1
ii. NOT (12345678)

i. i. AND REG^1, z1234
ii. AND REG^3, #000000F0
iii. AND REG^1, (FFFFFFF1)

j. i. ASR 3, REG^{3F}
ii. ASR 5, 10000000
iii. ASR 5, (REG^1)

k. i. ROR 8, (REG^1)
ii. ROR 31, REG^3

l. i. JMP (REG[3])
ii. JMP z4567
iii. JM6 z4567
iv. JM7 z4567

m. i. CLL (REG[1])
ii. CL4 22223333

n. RTS

3. For each part, a set of consecutive instructions are executed. Use the initial values of Exercise 2. Indicate what changes are made at the end of each set.

a. ADD REG[1], REG[7]
ADC REG[2], #00004567

*b. PSH REG[1]
PSH REG[2]
POP REG[3]
PSH REG[1]
POP REG[4]
POP REG[5]
POP REG[6]

c. CLL 1000000
PSH REG[2]
POP REG[6]
RTS

4. Modify the DDL of MODEL to add two new addressing types: modes 1010 and 1011:

a. Indirect, then indexed by the register specified by $IR_{26:31}$
b. Indexed by the register specified by $IR_{26:31}$, then indirect

5. Design a different version of multiple indirect from the one in Solved Problem 7. The address will be Page zero and all indirect addresses will also be Page zero.

6. Solved Problem 8 only provides a 32-bit product. Modify the design so that it will produce a 64-bit product, storing the answer in a register pair. (See Solved Problem 9.) Only the **z** bit should be changed.

7. a. Revise the solution to Solved Problem 9 to allow the second operand to come from a register pair.
b. Also, allow the second operand to be immediate. In the case of low-order half stored in the second word.

8. *a. We wish to provide several double-precision instructions. They work on data from a register pair and from another register pair, memory, or immediate (as described in Exercise 7b). Revise steps 1 to 18 to, for double-precision, fetch the data specified by the address field into `WORK` for the least significant part and into `WORK2` for the most significant half. Assume that only such instructions have a `1` in bit 2 (IR_2) of the op-code. (It is still to work as before for single-precision, $IR_2 = 0$.)

b. Redo Solved Problem 9 to accommodate this change.

9. Show the DDL for an instruction to count the number of 1's in the word specified by the address field, storing the answer in the register specified by RN.

10. Using the controller design from Appendix A, show a block diagram of the hard-wired controller to implement the shifts and rotates (starting at step 43), assuming the decoder at step 20 reaches step 43.

11. Consider the multiplication instruction of Solved Problem 8.

a. How long does the execution take, as a function of the number of 1's in the multiplier, *n*?

b. If registers could be cleared without using the bus and COUNT could be incremented without using the bus, what steps could be undelayed? How much improvement in speed would result?

c. Add a branch that quits the loop once the multiplier reaches 0.

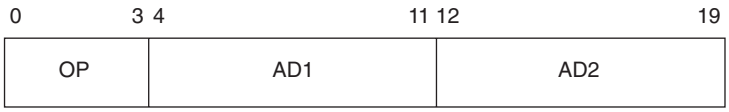
d. Compare the speed of the three approaches for a multiplier of 0000 0000 0001 1010 0001 0011 0001 1000

***12.** You are designing parts of a computer with a memory of 2^{25} 20-bit words. Instructions require 1, 2, or 3 words depending on the address modes. There is a bus structure similar to that of MODEL. Memory signals are the same as in MODEL, **but** reads and writes take two clocks. The first two steps of the DDL are

```
1. ADIN = PC; read = 1 ;← ADD25[1^25;
   PC].

2. IR ← DATA.
```

Note: There are two adders: a 25-bit adder (with no carry-in or carry-out) for addresses (as used in step 1), and a 20-bit adder with carry-in and carry-out for all arithmetic. There are thirty-two 20-bit registers.

The first word of the instruction has the following format:



Some instructions (OP beginning with 0) have two addresses; others have only 1 (using only AD1 for address computation).

The following address types are available. Those beginning with a 0 are complete in the AD field; those beginning with a 1 require an extra word. If both addresses require a second word, the word associated with AD2 comes first.

000xxxxx	Register Addressing
001xxxxx	Register indirect (Page zero)
010xxxxx	Register indirect with auto-pre-decrementing, where xxxxx represents a register number
100xxxxx	Direct, where xxxxx are the first 5 bits of the address
101xxxxx	Indirect, where xxxxx are the first 5 bits of the address and the indirect address
110xxxxx	AD1: Relative (second word sign-extended) AD2: Immediate (three words)
111xxxxx	unused

The following are the instructions included in the problem (with flags affected shown):

0000	Add	*z*, *s*
0001	Add double-precision	*z*, *s*
0010	Subtract	*z*, *s*
0011	Subtract double-precision*	*z*, *s*
0100	Compare	*z*, *s*
0101	AND	*z*,, *s*
0110	Move (from AD2 to AD1)	*z*,, *s*
0111	Move block of 16 words, not for register or immediate addressing	
1000	Increment	*z*, *s*

*For double-precision, the low-order half is at an even address or register number, and the high-order half is at an odd address or register number. We will assume that the programmer always enters an even address.

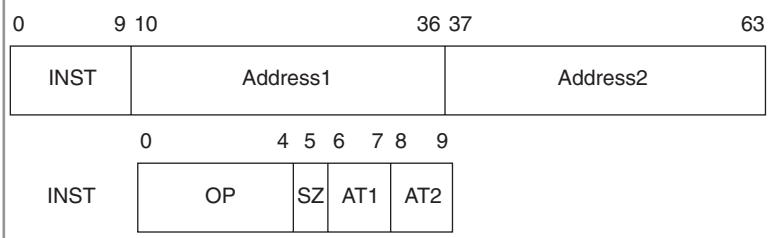
1001	Decrement	z, s
1010	Jump to subroutine (unconditional); return address stored in register specified by right 5 bits of AD2	
1011	Jump—condition specified by right 3 bits of AD2	
	0XX	unconditional
	100	z
	101	z'
	110	s
	111	s'

a. Write the DDL description of the machine. You may assume that only legal instructions occur. Make steps undelayed where possible to make it run reasonably fast. **Annotate your DDL!**

b. Produce a set of timing tables for all the instructions and all address types.

13. Design a computer with 128 Mwords of memory that operates on data of 32 and 64 bits. Memory is word-addressable, that is, addresses are 27 bits. The memory bus is 64 bits, and thus 2 consecutive words are accessed at once. These are always aligned so that the first 26 bits of the address of all instructions and 64-bit data is the same. All data comes from memory, and all results go to memory. Thus, there are no user registers comparable to REG in MODEL. There is a bus structure similar to that of MODEL.

All instructions require 64 bits. Thus, the PC need only be 26 bits (since all instruction addresses end in 0). The instruction format is as follows (where the details of the first 10 bits are specified afterward):



The result always goes to the location specified by Address1. For those instructions requiring one operand, its location is specified by Address2; for those requiring two operands, the first comes from the location specified by Address1, and the second from the

location specified by Address2. The size of the data is specified by the SZ field as follows:

0	32 bits	word (W)
1	64 bits	double word (DW)

You may assume that all double-word addresses end in 0. Or, if you prefer, you may ignore the last bit.

The following address types are allowed:

00	Direct
01	Indirect*
10	Indirect with auto-post-incrementing*,†
11	Relative (to the first word of the next instruction) for Address1 Immediate for Address2 (sign-extended)

Only the following instructions are to be implemented:

00010	ADD‡
00011	SUB(tract)‡
00100	AND
00101	OR
10000	MOV(e)
1010*x*	Convert from the size specified by SZ to the other size. If the conversion is to a smaller size, then overflow may occur and the appropriate flags should be set. When making numbers longer, sign-extend them.
11 *xyz*	JMP

Jump is available both conditionally and unconditionally. Address1 is the address of the next instruction. Bits *xyz* contain the condition code, as follows:

000	Number specified by Address2 is 0.
001	Number specified by Address2 is nonzero.
010	Number specified by Address2 is negative.

*Indirect addresses occupy the right 27 bits of a word.

†*Caution*: You add 1 for W and 2 for DW.

‡There are two overflow flags: one to indicate signed overflow (*v*), and one to indicate unsigned overflow (*c*). They can be tested by the jump instruction.

011	Number specified by Address2 is greater than or equal to 0.
100	Signed overflow (v)
101	Unsigned overflow (c)
110	Unconditional
111	CLL* (subroutine, unconditional)

All others unused

The CPU has a 64-bit adder that is used for all arithmetic operations including incrementing. Word operations use the right 32 bits. Address computation is also done using the right bits of that adder. When words (32-bit data) are read from or written to memory, they may appear on either the first 32 bits of `DATA` or the last 32 bits of `DATA`. You must account for that. There are two write signals. Both must be made 1 to write 64-bit words. It has memory connections `ADIN[0:26], DATA[0:63],` **`read, write0`** (for even addresses) and **`write1`**.

Example—to read word data, the address of which is in EA

```
ADIN = EA0:25; read = 1;
EA26: WORK ← 00000000, DATA0:31;
EA26: WORK ← 00000000, DATA32:63.
```

a. Write a complete DDL description of a hard-wired controller for this machine (including undelayed steps). You may assume that unused codes and illegal combinations do not happen. You may add whatever internal registers you need (such as `WORK` in MODEL). You are to make this machine run reasonably fast (that is, take advantage of undelayed steps wherever possible), without writing very complex code. **You must annotate your solution**—at least to show where the steps for each op-code and each addressing type begin.

b. Compute the timing for each instruction and addressing type. (The timing should be the same for both sizes of data; but if it isn't, you must include that in your computation.) **You must show a diagram or a listing for the steps executed for each.** Display your results in a readable manner. You need not show a table as we did for MODEL; it would require three dimensions. Rather, you can show the timing as composed of the sum of

*The return address is stored in the last word of memory (that is, addresses 7FFFFF). No provision is made to nest subroutines. You do not have to check.

three or four parts (for example, instruction fetch plus address 1 plus address 2 plus execution), with a table for each part. Just make sure that it is clear how you compute the timing for any instruction.

10.8 CHAPTER TEST (75 MINUTES)

1. (25) For the following values in MODEL:

PC = 12000122
Bits 16:31 of this instruction word are 9402
REG^3 = 98765432
M[00000000] = FF000011
M[12121212] = 00000000

Show the changes to registers, flag bits, and memory locations for each of the following instructions. Also specify the number of memory references to fetch and execute each instruction.

a. LOD REG^2, 12121212
b. STO REG^3, z4567
c. ADD REG^3, #80112233
d. AND REG^3, [12121212]
e. JMP @9000

2. (25) I wish to create a new instruction for MODEL. It only works for those addressing types that produce a memory address in EA, but you need not modify the addressing section to check for that. You will need an extra register, TEMP (if you don't want to change any other registers or memory locations).

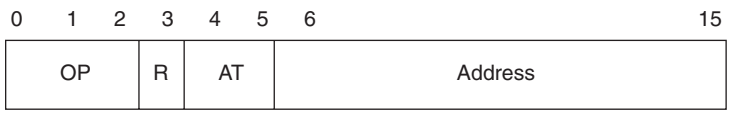
This instruction, SWP, compares the unsigned number in the memory location pointed to by EA with the unsigned number in the location following that. If the second number is greater, it swaps the two numbers; otherwise, it does nothing. Write the DDL to implement this instruction beginning at step 60.

Examples: SWP 00001234

Before:	00001234: 7 00001235: 8	After:	00001234: 8 00001235: 7
Before:	00001234: 8 00001235: 7	After:	00001234: 8 00001235: 7

3. (50) You are involved in the design of a small specialized computer, SMALL. It has a memory of 2^{16} 16-bit words. Instructions may only be executed from the first 2^{10} words; thus the PC need only be 10 bits and the Address part of the instruction is large enough to hold a complete address for the jump instructions. The machine has two registers, REG^0 and REG^1. There are no flag bits. The adder adds two 16-bit numbers and produces a 16-bit result. The bus structure is similar to MODEL. The instruction format is as follows:

The AT field specifies the addressing type, as follows:



00 Page zero (that is, 6 leading 0's)
01 Unused
10 Page zero indirect
11 Immediate (only allowed for first four OP codes, zero-extended)

The OP field specifies one of eight instructions, six of which are defined as follows:

000 Load register from memory (or immediate)
001 AND number from memory (or immediate) to register
010 Add number from memory (or immediate) to register
011 unused
100 Store number from register into memory
101 unused
110 Jump (to ADDRESS) condition specified by AT*

- 00 always
- 01 REG = 0
- 10 REG > 0
- 11 REG < 0

* Only Page zero addressing is allowed for the two jump instructions.

111 DJZ*: Decrement register (specified by R) and jump (to Address) if register had been (before decrementing) 0 (AT is ignored)

The R bit specifies which register.

Write the DDL code for this machine. Assume that the unused codes and improper combinations (such as store immediate) never occur.

* Only Page zero addressing is allowed for the two jump instructions.