5. CPU Implementation

5.1 Defining an Instruction Set

Before going into the details of the datapath we must define the Instruction Set. There are some major decisions that are to be made:

- the **data path width**: do we design our Instruction Set for an eight bit machine (not very probable), for a 32 bit one or maybe for a 64 bit, top of the art machine? We choose a 32 bit architecture, mainly because the market is, at this moment dominated by these machines, but also because the major tradeoffs can be easily discussed. The design of a wider architecture, say 64 bit, poses special problems (like multiple instruction streams- superscalar architectures), problems that are beyond the scope of this class.

- **support for pipelining**: even if we don't discus about pipelining it should be clear by now that a "clean" Instruction Set eases the hardware implementation; in particular it is extremely important if we settle for an instruction set in which all instructions execute in the same number of clock cycles (this is paramount for pipelining), or for some bushy instruction set, full of addressing modes, and all kind of nice instructions.

We follow the current trends and we choose to have a simple instruction set, whose main aim is to permit a simple hardware implementation; we shall select those instructions and addressing modes that allow instructions to run in the same number of clock cycles. We implicitly settle for a fixed length instruction format: variable length instructions mean variable number of clock cycles for fetch.

We have a 32-bit data path; instructions will be also 32 bit wide. Clearly it does not make sense to have narrower instructions as long as we want to fit so many things in a single instruction: opcode, source and destination operand(s), immediate value, offset/displacement and maybe others. On the other hand going beyond 32 bits with a 32 bit architecture means more clock cycles to fetch instruction, more memory traffic, without any guarantee this won't offset the benefits of wider instructions.

- **number of addresses**: is this one, two, or three address architecture? The examples in the previous lessons should have convinced you that a one address machine doesn't look very exciting given the current technologies. To choose between two and three address is more difficult: those who advocate for two address machines say the three address ones much too often use the same register both as a source and as a destination thus behaving as two address instruction machines. On the other hand, those who claim three address machines are better say it is true that in many cases three address instructions behave like two address ones, but if three distinct operands are to be used then a two address machine must use two instructions to do the same job, and it uses also an extra register.The new designs are three address machines and the same will be ours.

- **register-register, register-memory or memory-memory**? Memory operands were important at a time when technology could not provide enough general purpose registers; it should not be understood from this that we no longer need instructions that take operands from memory: this is powerful and useful in very many situations. But again there is the performance aspect we must consider: memory operands make instructions slower; and if we want instructions that work both register-register and register-memory or memory-memory then we must also accept a variable number of clock cycles for instructions to execute. All CPUs designed in the last years are register-register; so will be the one we will be discussing about.

- the **number of registers** our architecture will have, a too small number of registers slows down the machine because variables have to be often swapped between registers and memory. A too small number of registers give hard times to the compiler writer also. On the other hand, a too large number of registers could mean valuable silicon area which is not very well used if some registers are “idle”, i.e. they don't hold some valuable information. Moreover the more registers we choose to have the more bits will be necessary in the instruction to specify them. Because the instructions are fixed size length (word) more bits for specifying registers mean less bits for immediate/offset/displacement field, thus negatively affecting the performance of instructions mainly of branches.

We follow again the current trend, and choose to have 32 general purpose registers. It is to be mentioned here that the designers of the DEC's ALPHA architecture (a 64 bit one) have also settled for 32 registers, for the main reason that more registers would not improve significantly the performance.

A closely related problem is that of the floating point registers: do we include a special set of registers for the floating point operations or not? As a matter of fact most of the CPUs designer in the last years do include a special set of floating point registers, meant to be used by the floating point instructions. The main problem with the floating point operations is that they don't fit nicely in a simple instruction set; while an integer addition or multiplication can be performed in a single clock cycle (assume) that hardware is provided for this), a floating point multiplication or division may take several (5-10). We shall focus our attention on the integer part of the CPU, and ignore, for the time being the floating point part.

To conclude this introductory part, here are the main features of our architecture:

- 32-bit data-path
- register-register
- 32-bit wide instructions
- 32 general purpose registers
- instructions executing in the small number of clock cycles.

We did not consider all aspects of deciding an architecture; we will do some while we decide what instructions should be included in the Instruction Set. As a very simple example, should the memory accessed by this CPU, be byte or word addressable? If we decide to make it byte addressable then we can include a load_byte instruction in the instruction set; otherwise we must simulate byte instructions using other instructions in the set.

5.2 The Instruction Set

Addressing Modes

At a first glance the more addressing modes we provide, the more convenient will be to use the instruction set; the question is, convenient for programmers in assembly language or for the compiler? As most of programmers no longer use assembly language, the Instruction Set we design must be convenient for the compiler. This is to say, the instruction set has to be designed in such a way to ease the writing a compiler for the new machine. If we provide ten different addressing modes, then a big amount of time of the compiler writing team will be spent in trying to figure out what the best addressing mode is for a sequence of instructions. And when it comes to optimizations thing get even worse, because a large number of possible sequences that yield the same final result, have to be analyzed to decide which one is best.

Our machine will have only three addressing modes:

- register
- immediate
- base register + offset.

Obviously, any addressing mode can be synthesized using the base-offset (or base_displacement if you prefer) addressing mode, and the sequences of instructions required to simulate other addressing modes are simple and clear. Thus the compiler development job is greatly simplified.

Operations

We must provide instructions for the three basis classes: ALU operations, loads and stores, branches and jumps.

ALU instructions are register-register (clearly because we have a register-register machine). Before deciding what operations we want to include in the Instruction set, we must decide if we allow immediate operands in ALU instructions.

```
add r1, r2, 3
```

is clearly an easy way to add the immediate 3 to the content of register r2, and to store the result in register r1. Without immediate operands the sequence would be:

```
lw  r3, (r4)
add r1, r2, r3
```

which assumes that the number 3 is stored at the address in r4. This is not a problem because the compiler usually decides that literals are kept in a dedicated memory area (the literal pool) where they are accessible throughout the live of the program; as such the address of any literal is known at compile time, and generating the above sequence of two instructions is easy. What does this pay for us?

- if we accept immediate operands in instructions then we must somehow transform the immediate into a 32 bit number; remember that the immediate occupies only a field in the instruction, it is not 32 bit wide; there will be some hardware complications for this.

- if we do not accept immediate in instructions (the second sequence of code) then we don't have to worry about possible hardware complications at this point; it the compiler's job to properly represent the numbers. The price paid is a sequence of two instructions instead of one, and the use of a register to hold the number before the actual arithmetic operation is performed.

As a matter of fact literal integers appear quite often in programs (think about comparisons in the language you are using, Pascal, C, whatever). In these conditions we'll allow immediate operands in our ALU instructions.

To keep things very simple we include only those operations that are used the most, taking care to provide enough support for the simulation of others.

Here they are and don't forget we only discuss about integers.

Operation	Mne-monic	Example	Meaning
Addition	add	add r1, r2, r3	r1 ⟵ r2 + r3
	addi	addi r1, r2, 5	r1 ⟵ r2 + 5
Subtraction	sub	sub r1, r2, r3	r1 ⟵ r2 - r3
	subi	subi r1, r2, 5	r1 ⟵ r2 - 5
Multiply	mul	mul r1, r2, r3	r1 ⟵ r2 * r3
	muli	muli r1, r2, 5	r1 ⟵ r2 * 5
Division	div	div r1, r2, r3	r1 ⟵ r2 / r3
	divi	divi r1, r2, 5	r1 ⟵ r2 / 5
Logical OR	or	or r1, r2, r3	r1 ⟵ r2 or r3 (bitwise or)
	ori	ori r1, r2, 5	r1 ⟵ r2 or 5 (bitwise or)
Logical AND	and	and r1, r2, r3	r1 ⟵ r2 and r3 (bitwise and)
	andi	andi r1, r2, 5	r1 ⟵ r2 and 5 (bitwise and)
Logical XOR	xor	xor r1, r2, r3	r1 ⟵ r2 xor r3 (bitwise xor)
	xori	xori r1, r2, 5	r1 ⟵ r2 xor 5 (bitwise xor)

Certainly ALU instructions must be able to operate between any registers in the register file (the 32 registers our machine have). As it is always the case with arithmetic operations, a representation for the numbers has to be chosen. It is important because the hardware looks different for different representations.The two's complement representation for integers certainly is the most widespread.This is all by now; as we'll see soon some other operations must be included in the group of ALU Instructions.

Loads and Stores load and store any register. We must now decide what kind of loads/stores we want: word only or maybe word and half-word and byte. This is a major decision because it influences not only the instruction set but also the memory subsystem. If we decide our load/store instructions will handle only words then we don't have to be concerned with memory alignment problems. The price we pay for this is a difficult manipulation of characters (represented as bytes).

Suppose again we want to keep things very simple and we decide for word only operations. Then we have the following:

Operation	Mnemonic	Example	Meaning
Load	lw	lw r1, (r2)	r1 ← M[r2]
Store	sw	sw r1, 8(r2)	M[r2+8] ← r1

Even though we have only two instructions, we must keep in mind that their number would be larger in the case byte and half-word load/store operations were allowed.

Branches and jumps. Let's start with a question: do we want or can we have an instruction like

```
jump        label
```

where label is an absolute address? The answer is we want (because it is the most natural way to specify the change in the flow of control) but we can not as long as we insist in having fixed size instructions. With a 32-bit instruction size and a 6 bit opcode field there are only 26 bits that could be used as an immediate address, much too little. Actually we have already discussed about this and possible solutions to this problem:

- jump relative to the PC: use a signed offset to compute the target address by adding it to the PC;
- jump through register: the jump instruction specifies a register that contains the target address.

Operation	Mnemonic	Example	Meaning
Jump	j	jname	PC ← name
Jump Register	jr	jrr1	PC ← r1

Note that the syntax for the plain jump can be misleading: name is a symbolic name and not an absolute address; moreover the instruction that is labeled name must be within the range accessible by the displacement a jump instruction can provide.

Branches specify a conditional control transfer, and we must decide how conditions are set and tested and how the target address is computed. If we start with the latter, the answer is easy: target address cannot be an absolute address, because we can not accommodate absolute addresses in our fixed size instructions. We can adopt the same policy as for jumps with the following differences:

- the offset will be shorter as a branch instruction has to specify what has to be tested;

- a branch through register may raise some hardware problems because first a comparison must be done and then a target address must be computed; probably this is a real concern if we decide to pipeline our machine. Let's keep both kind of branches and see if we get some problems in implementing them.

Deciding what conditions to test and how the conditions are set by previous instructions is not easy. Here are the major ways for evaluating branch conditions:

- **Condition Code** (CC): special bits in a flags register are set by ALU operations or explicitly using special instructions; examples include Z (the result of the last arithmetic operation was zero), C (there was a carry at the last operation), V (there was an overflow at the last operation) etc. The main advantage of CC is that, in most cases, they don't require extra work/time to get set. The disadvantage is that condition codes impose restrictions in reordering instructions; you may need to reorder instructions when you write an assembly program or, more often, reordering appears in compilers as a way to optimize the code generated.

- **Condition Register** (CR): the result of a comparison is placed in some arbitrary register. The advantage is that this is very simple, though it requires an explicit instruction to set the register, and it uses a register to hold the result of the comparison. It simplifies however the compiler's job in that any instructions can be inserted between set instruction and a branch as long as these instructions don't have as destination the condition register.

- **Compare and branch:** compare is a part of the branch, as a as a result there is only one instruction that compares and branch instead of two as was the case with CR. There is a problem however in that there might be too much information to be encoded in the instruction and too much work to be performed in a fixed amount of clock cycles (if we care about).

With all the above in mind let's try to decide how our branches should look like:

- compare and branch is not very appealing for a register-register machine with fixed instruction length: the instruction should specify two registers (10 bit in our machine), the comparison that has to be performed (greater_than, equal, etc.) which means 3 more bits, all this besides the opcode (say 6 bits), as a result the offset can be only

32 - 6 - 10 - 3 = 13

bits wide. On the other hand performing a comparison and the computation of a new address can prove to be too much if we want to make instructions run in the same amount of clock cycles (a small amount of course as we have already discussed).

- using condition codes (CC) is also problematic due to the problems it poses to compiler writer. Sure, these problems can be overcame, many very successful machines (as the VAXs or the Intel x86) use condition codes and yet very efficient compilers have been written for those machines as targets. But behind the scenes, there was, more or less, a hard work in trying to figure out what sequences of code can be replaced with what without changing anything in the machine's status.

- most of the machines that emerged in the last years use the Condition Register method for branches. This is the way we will choose our instructions.

Operation	Mnemonic	Example	Meaning
Branch	beqz	beqzr1, name	if (r1 == 0) PC $\leftarrow$ name;
	bnez	bnezr1, name	if (r1!= 0) PC $\leftarrow$ name;

Note that name represents a symbolic label and not an absolute address. When code is generated it will be translated as a signed offset that is added to the PC to get the target of the branch when it is taken. If there are k bits for the offset i an instruction then name must respect the following condition:

$(PC+4)-2^k <= name < (PC+4)+2^k$

The content of a register a branch uses for test can be some value left there by some computation, or it cat be the result of a set operation; we have, at this moment to update our ALU Instruction pool with instructions that compare two registers (or a register and a immediate) and set the content of the destination register: if the condition is true place a 1 in the destination, else place a zero (the positive logic convention for True and False respectively). This instructions are of the form:

```
sxx  d, s1, s2    d is the destination register;
                  s1, s2 are the source registers;
sxxi d, s1, immediate
```

where xx in the above mnemonics stands for one of the following:

xx	Example	Meaning
eq	seq r1, r2, r3	if (r2 == r3) r1 ← 1; else r1 ← 0;
ne	snei r1, r2, 7	if (r2 != 7) r1 ← 1; else r1 ← 0;
lt	slt r23, r2, r27	if (r2 < r27) r23 ← 1; else r23 ← 0;
le	slei r23, r2, 0	if(r2 < 0) r23 ← 1; else r23 ← 0;
gt	sgti r11, r12, 3	if (r12 > 3) r11 ← 1; else r11 ← 0;
ge	sge r11, r12, r13	if (r12 >= r13) r11 ← 1; else r11 ← 0;

There is one more problem that has to be resolved, calls to subroutines and returns. They are like jumps with the following differences:

- at a call the return address must be saved in stack
- the return address is not known at compile time; from the two flavors of jumps, the return is like jump through register.

Because they resemble so much to jumps we'll choose similar names:

Operation	Mnemonic	Example	Meaning
Jump And Link	jal	jalname	r31 ← PC+4; PC ← name;
Jump And Link Register	jalr	jalrr7	r31 ← PC+4; PC ← r7;

There is no need for a special return instruction; we have already one, jump register (jr) which does the job if the register contains the proper return address. The question you may ask at this moment is "why do you save the return address (PC+4) in register r31 and not directly into the stack?" and it makes very much sense. After all, if we place the return address in r31 we must explicitly transfer it into the stack in the subroutine, while an implicit push into the stack would save us a few lines of code. To answer this question you must first recall one of the basic principles of a computer designer which is:

make the common case fast

When we enter a subroutine the content of r31, i.e. the return address from that subroutine, must be saved only if there is another call inside that subroutine (nested subroutines). As long as most of the subroutine calls are of the form:

```
name:               # start subroutine
...................# no call statement (no jal or jalr)
jr    r31          # return to caller
```

there is no need to save the return address into the memory, thus saving two memory accesses (which is not only a matter of time saving but also one of reducing the memory traffic).

Example 5.1 IMPLEMENTATION OF A SEQUENCE:

Show how to implement a call return sequence if the subroutine must save the return address into the stack.

Answer:

```
      ..................
sub1: subi  sp, sp, 4
      sw    r31, (sp)    # push return address into the stack
      ..................# the subroutine's body
      lw    r31, (sp)    # pop from stack the return address
      addi  sp, sp, 4
      jr    r31          # return
      ..................
      ..................
      jal   sub1         # call sub1
      ..................
```

Note: sp denotes the Stack-Pointer, a register from the 32 our machine have, dedicated to handling the run-time stack.

Instruction format

We can now more precisely indicate the format of instructions. There are three different formats as can be seen below:

I-type

6	5	5	16
Opcode	s1	d	immediate

R-type

6	5	5	5	11
Opcode	s1	s2	d	func

J-type

6	26
Opcode	offset

The I-type instructions encode:

- ALU operations with immediate d ← s1 op immediate
- Load and Store
- branches (d is unused)
- jr and jalr (d and immediate fields are unused).

The R-type instructions encode:

- all register-register ALU operations: d ← s1 func s2

The j-type instructions encode:

- j and jal instructions;

There are some points where you may raise justified questions:

- why don't we use the opcode field to indicate the operation in the R-type instructions (and use instead the func field)?

- why don't we have a special instruction format for branches? The 5 bits of the d field are unused in branches, and, maybe, they could be used to have a larger offset (the immediate field is used as an offset)

Let's try to sketch answers to these questions, even if a full understanding of the problem
requires a discussion about pipelining, which is beyond the scope of this course.

- one of the major concerns in implementing the CPU is how to reduce the clock cycle thus improving performance; while technology has played the major role in the evolution of computers, architectural innovations and design refinements have their fair share in the process. Suppose there are several logical steps in the execution of an instruction (fetch, decode, ALU...); when it comes to implementation it turns out that they require different amounts of time to execute: we want to reduce all execution times until they get equal (or at least very close) to the minimum execution time from all logical steps. It so happens that the step following the instruction fetch, decode and register fetch, may be required to more than that in a pipelined implementation: more precisely in this single step the instruction must be decoded (decode the opcode field), and then, if it is a branch a comparison must be performed to find out if the branch is taken or not; if it is taken the modify the content of program counter, PC ← target_address (the target address can be computed in parallel with the decoding, in many cases the result makes no sense, but this is not a problem). It follows that the decoding time should be reduced as much as possible. This gives the answer to the first question; the less we encode in a binary field the easier the decoding, at limit no decoding at all! this is also true for the func field, the required decoding for this field should be very little.

- the answer to the second question is simpler: both for immediate operands and for offsets there must be some hardware to transform the immediate/offset into a full 32 bit integer that can then be operated; to simplify hardware only one such device is provided, namely for the shortest of the two possible. It would be nice to have larger offsets, but the price is not worth.

5.3 Executing an Instruction

Figure 2.2 presents the schematic diagram of an ordinary CPU. We shall use that block diagram to see what are the steps in the execution of an instruction.

Two steps are common for all instructions:

1. Instruction fetch

`MAR ← PC;` Place the content of PC in the Memory address Register.
`IR ← M[MAR]`;The instruction is brought into Instruction Register.

Note that these are logical steps. Our implementation should reflect the fact that the memory access may take several clock cycles.

2. Instruction decode and register fetch

`PC ← PC+4`

Operands specified by the s1 and s2 fields in the instruction are retrieved in the register-file and brought at the A and B outputs. PC is updated to point to the next instruction in memory. Instruction decoding proceeds in parallel with register fetch: this is possible because the source fields in the instruction are always in the same positions, and as a result there is no need to wait until the instruction is decoded to decide where are the operand to be taken from. The immediate values also belong to a fixed field, and the sign-extension of immediate operands can be done in this step.

Obviously registers specified by s1 and s2 fields will always be accessed in the register file even if the instruction does not need register operands, as is the case with a jump (j); this does not hurt. Note in this context that we need multiple port register-file, with two read ports and a write port: this should allow us to read two different registers and to write a third one in a single clock cycle (of course the three registers may be the same).

ALU is used to compute the new value of PC; for this to happen the control unit must generate the proper control signals to PC (to place the content on one of the busses Op1 or Op2) and ALU. The constant 4 might come from a small table of constants, or the ALU can be design in such a way to directly provide a +4 operation. After the instruction has been decoded there are some steps which differ from one instruction to another. We shall present these steps for different kinds of instructions.

ALU Instructions

3. ALU operation and write back

```
ALUoutput ◄— A op B    (or ALUoutput ◄— A op immediate)
d         ◄— ALUoutput;
```

Writes the result of the operation into the register-file (write back), in the destination register specified by the field d in the instruction. The write back happens at the end of the clock cycle, after the computation has been done and the output lines of ALU are stable.
This step concludes the ALU instructions.

Branch Instructions

3. Target address computation

```
ALUoutput ◄—  PC +offset; cond ◄— A op 0;
```

offset denotes the sign extension of the immediate field in the instruction, i.e. bits 15-0 in IR. The comparison against zero is also done in this step.

op	What happens
eq	if (A ==0) cond $\leftarrow$ 1; else cond $\leftarrow$ 0;
ne	if (A!= 0) cond $\leftarrow$ 1; else cond $\leftarrow$ 0;

Condition cond will be tested in the next step to see if the branch is taken or not.

4. Test and set new PC

if (cond) `PC` $\leftarrow$ `ALUoutput`;If the condition cond is true the branch is taken and the content of PC is replaced by the target address (calculated in step 3). Otherwise the content of PC is left unchanged.
This concludes the execution of a branch instruction.

Jump Instructions (simple jump)

3. Target address computation

```
ALUoutput ◄— PC + offset;PC ◄— ALUoutput;
```

offset denotes the sign extension of the offset field in the instruction, i.e. bits 25-0 in the Instruction Register. At the end of this clock cycle the output of ALU is stored into PC; this one must be edge triggered for correct operation. This concludes a simple jump.

Jump and Link Instructions

3. Save return address

```
r31 ◄— PC;
```

PC is saved into r31. PC has been incremented by for in step 2 such that the saved value is the address of the next (in order they are listed) instruction.

4. Target address computation

```
ALUoutput ◄— PC + offset;PC ◄— ALUoutput;
```

offset denotes the sign extension of the offset field in the instruction, i.e. bits 25-0 in the Instruction Register. At the end of this clock cycle the output of ALU is stored into PC; this one must be edge triggered for correct operation. This concludes a Jump and Link instruction.

Load/Store Instructions

3. Memory reference

```
ALUoutput ◄— A + offset;
MAR       ◄— ALUoutput;
```

offset denotes the sign extension of the immediate field in the instruction, i.e. bits 15-0 in IR. If this step can be performed in a single cycle (ant it can) then the Memory Address Register (MAR) is loaded at the end of this cycle, when the outputs of ALU are stable. The coming cycle is a read/ write cycle; if it is a write cycle then data to be written should be already into the MDR. But this cannot be done in this cycle with the CPU organization in Figure 2.2: the only way from the outputs of register-file to MDR is through ALU, and ALU is busy at this moment computing the address. Loading MDR in the coming cycle involves some delicate timing (the load command to MDR must come after the outputs of ALU contain the proper data, but early enough such that a memory write cycle can still be performed), and it could be a better idea to provide a direct path from the outputs B of the register file, to the inputs of MDR. In these conditions we have to add at this step:

```
MDR  ◄—   B;
```

4. Memory read/write

```
M[MAR] ◄—  MDR;        in the case of a write;
MDR    ◄—  M[MAR];     in the case of a load;
```

The actual memory access is performed in this step; depending upon the speed of the memory, this step may take several cycles to complete. If it is a load then during the last clock cycle data coming from memory is stored in MDR.

This step concludes a Store instruction. In the case of a Load instruction there is one more step, writing back into the register-file.

5. Write back

```
d ◄— MDR;
```

d denotes the destination field in a I-type instruction. The content of MDR is written into the d register. This ends a Load instruction.

5.4 Hardwired Control

After the Instruction Set has been defined and the datapath has been designed, the next step is the design of the Control Unit. We didn't undertake a full effort to completely design the datapath; what we have in Figure 2.2, with some improvements suggested by the analysis of executing steps, is only a rough description of a datapath. A complete design should include more details and all control points; for example buffers that control access to internal buses Op1 and Op2 should appear on the figure, together with the name of signals that control them and their significance, i.e. if they are active low or high.

Such a detailed design is beyond the scope of this course; while the problems a designer faces in doing the whole specification and carrying the design are very interesting, it is a process in which technological details are very important, and, as such, a Digital Design class or a VLSI Design one are more appropriate.

Hardwired control means that the Control Unit is implemented as Finite State Machine with outputs going to every control point in datapath and in the outside environment (control signals for memory, bus operation, etc.), and inputs from all relevant parts of the structure (most of them come from Instruction Register but not only).The Finite State Machine is specified using a finite-state diagram. Every state in the diagram corresponds to a clock cycle. In every state input signals may be tested, and output signals may become active.

A first step in specifying the finite-state diagram is to draw a diagram in which macrooperations are performed, i.e.operations as they appear in the description of execution steps. For instance one state in the diagram could include the following statement:

```
PC  ←  PC+4;
```

This has to be further detailed in a later stage of the design: in order for this operation to be performed, several control signals must become active:

- open driver for PC content to get on the bus Op1;
- a signal (or more) to tell ALU it has to perform an addition with 4
- load enable signal for PC such that it will be loaded with the new value at the end of this clock cycle.

Figure 5.1 presents the part of the diagram, which corresponds to the first two steps in the execution of an instruction, the instruction fetch and the instruction decode. As it can be seen, after the content of PC is loaded into MAR, a memory cycle begins; the Control Unit repeatedly tests to see if the memory cycle has completed (the memory subsystem must provide such a signal to CPU). If the cycle has not completed (MemoryReady = No) then the Control Unit keep asserting the control lines necessary for the memory control (in state labeled Q0). After the instruction has been brought into IR (when MemoryReady = Yes) it gets into the state Q2 where the actual execution begins.

Figures 5.3 to 5.6 present the parts of the finite-state diagram describing different kinds of instructions. If we count them we find there are 40 states. Figure 5.7 presents the general view of a Finite State Machine, while Figure 5.8 presents A FSM in the context of a Control Unit.

To implement a FSM whose state-diagram has N states one needs:

$ns = \text{ceiling}(\log_2(N));$

bits to encode the state. In our case, to encode 40 states:

$ns = 6$

There are also 6 inputs from the IR, the opcode, and some other inputs from the func field of the IR, suppose 4 bits (16 different functions can be encoded wit 4 bits). As for the bits coming from the datapath, there those who indicate the relation between A and temp (see Figure 5.4), and those indicating the relation between A and zero (see Figure 5.5), a total of seven. At this we have to add some other conditions that might need to be tested, as the overflow from ALU. Suppose there are 10 inputs to the Control Unit from the datapath.

There are also some external bits and it is here to consider inputs from the memory (MemoryReady and PageFault, the latter in the case our system is supporting a virtual memory scheme), and maybe from peripheral devices, as one or more interrupt signals. This can not be detailed at this moment because the whole picture of the system should be available. Anyhow, suppose there are 4 input bits from the exterior.

As we said earlier, at this moment a detailed datapath design should have been done such that the total number of control signals is known. It is not unreasonable to assume this number is somewhere between 30 and 50. Suppose it is 40. With the numbers we have so far the Control Unit looks like in the Figure 5.9.

There are several ways to turn FSM specification into a real design:

1. Direct minimization

- MAR $\leftarrow$ PC; Place the content of PC in the Memory address Register.

- IR $\leftarrow$ M[MAR]; The instruction is brought into Instruction Register.

This method is suitable for very small designs, with only a couple of inputs. The combinatorial logic must implement 46 binary functions each function with 30 inputs (24+6). Note that the main problem is not the number of outputs (each new control signal is an output from the combinatorial circuit, and defines a new function), but the number of inputs, as every new input line doubles the complexity of the function. A truth table for a boolean k places function has:

$$2^k$$

rows. Functions have to be minimized and expressed in canonical form, usually as sum of product terms. After that the circuitry is laid down: for every product term there is a NAND gate, and for every function there is a big NAND gate that "adds" the product terms together.

The problem with this kind of implementation is the messy result; interconnecting all the gates is a challenging task. Once the design has been turned into silicon any changes are virtually impossible: if an error is discovered this usually implies a complete redesign of the Combinatorial logic. This is sad because it is precisely in the control unit where the most design errors are found.

2. ROM implementation

Any set of M functions, each of k places can be implemented using a ROM with k address lines and M outputs. For a combinatorial circuit with 30 inputs we need a ROM with 30 address lines! And we have 40 outputs, this means a 5GByte ROM. It is not quite what the technology permits at this moment. The problem with the ROM implementations is that a huge amount of logic is wasted; if a ROM has k inputs then all 2^k product terms (minterms) are computed, i.e. there is a gate with k inputs for every such minterm. Actually only a very small number of this product terms are being used to implement a function, the reason for this is that some combinations of input variables will never appear.

Minterms are at hand, we can easily modify the design to use another minterm(s), but this approach is very expensive if possible at all. This approach to the design of combinatorial logic is especially attractive for small designs (up to 16-20 inputs) in an early stage of the development when changes are frequent and ROM prove to be very flexible (actually the rewritable versions of ROM).

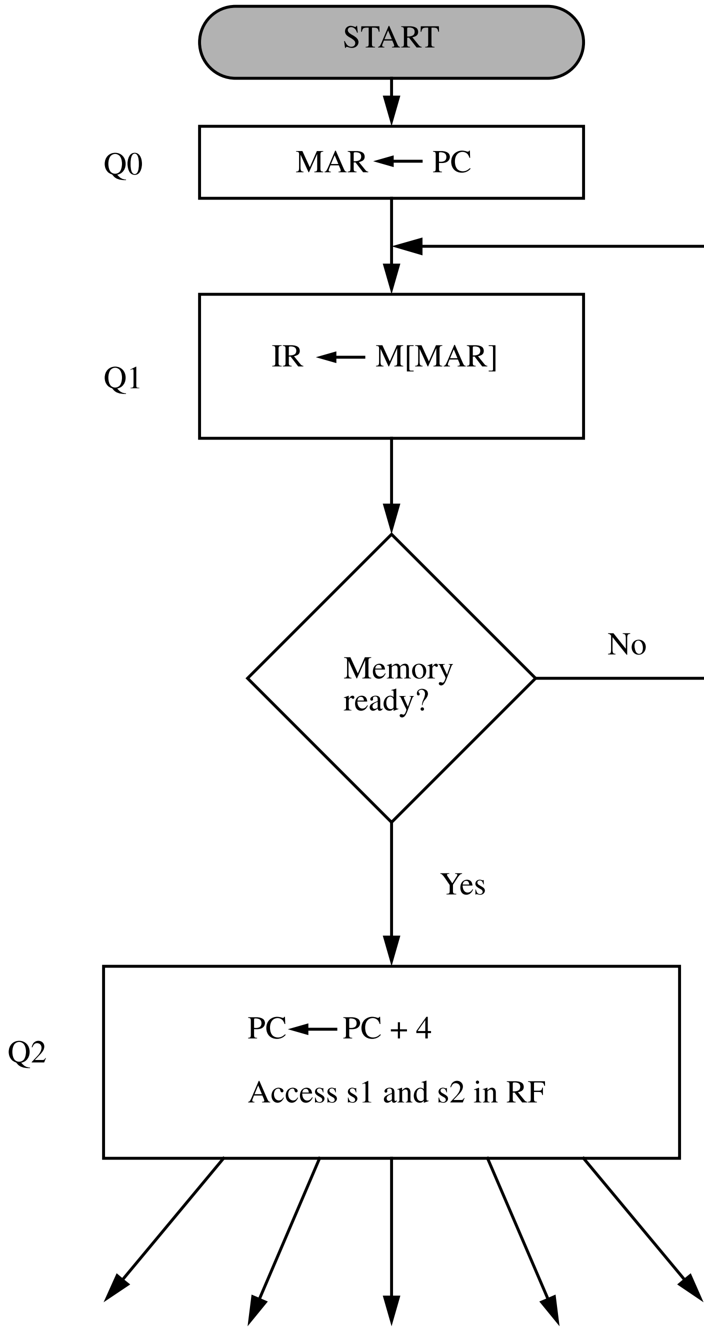

FIGURE 5.1 The states corresponding to the first two steps in the execution of an instruction, fetch and instruction decode.

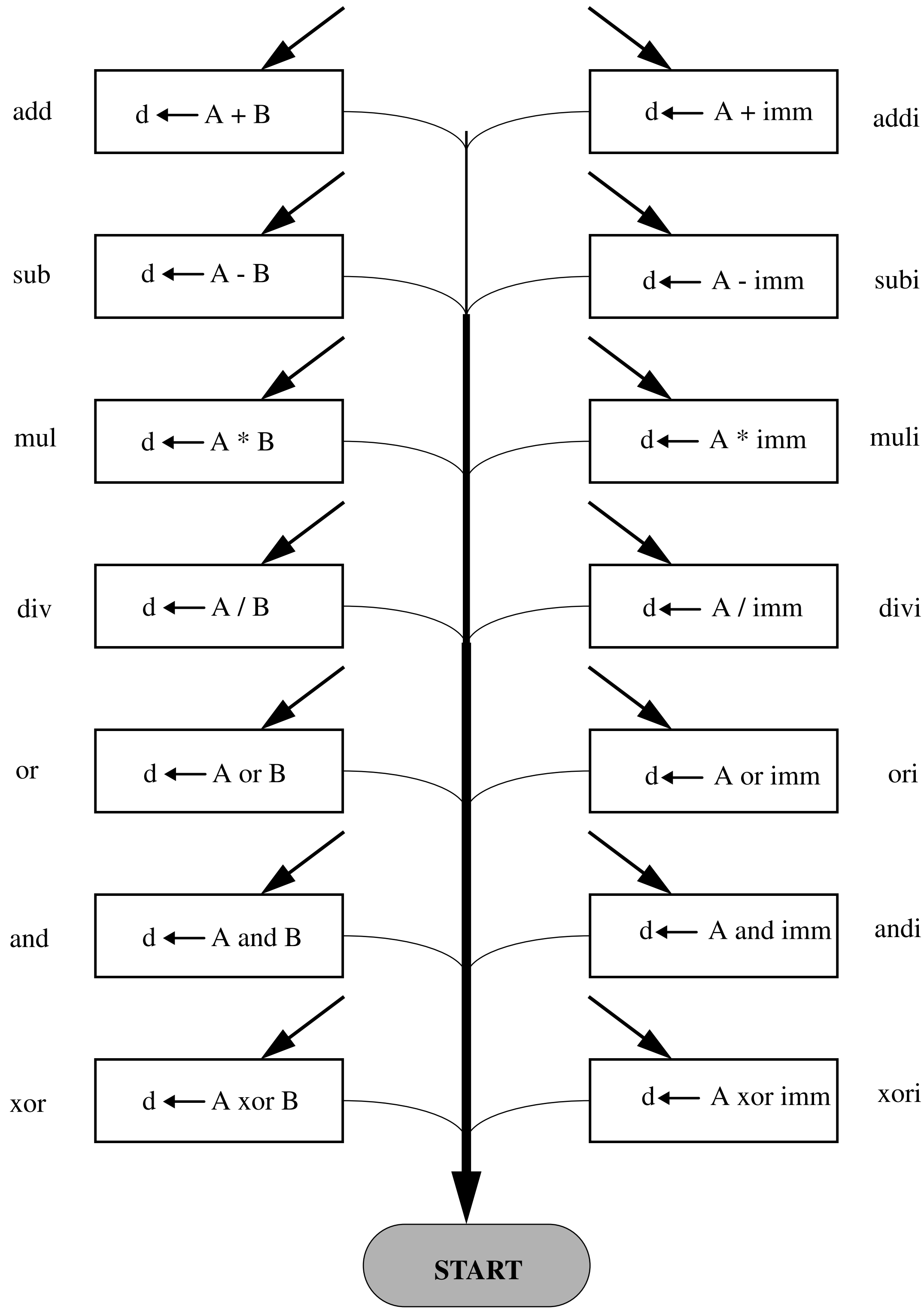

FIGURE 5.2 The states in the state-diagram corresponding to ALU operations.

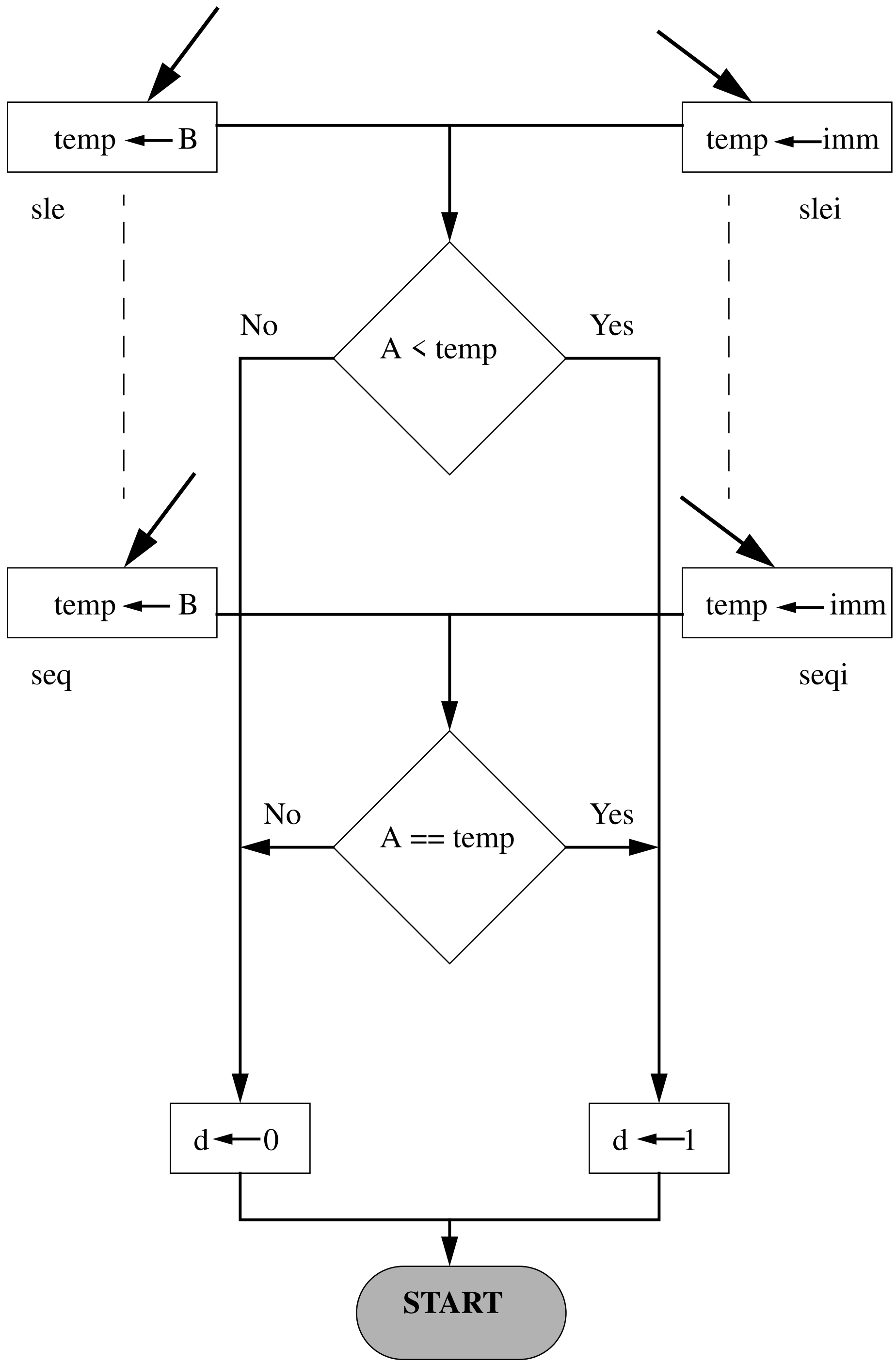

FIGURE 5.3 States in the state-diagram corresponding to some set operations. There are 14 states; only those for sle, seq, slei, and seqi are represented.

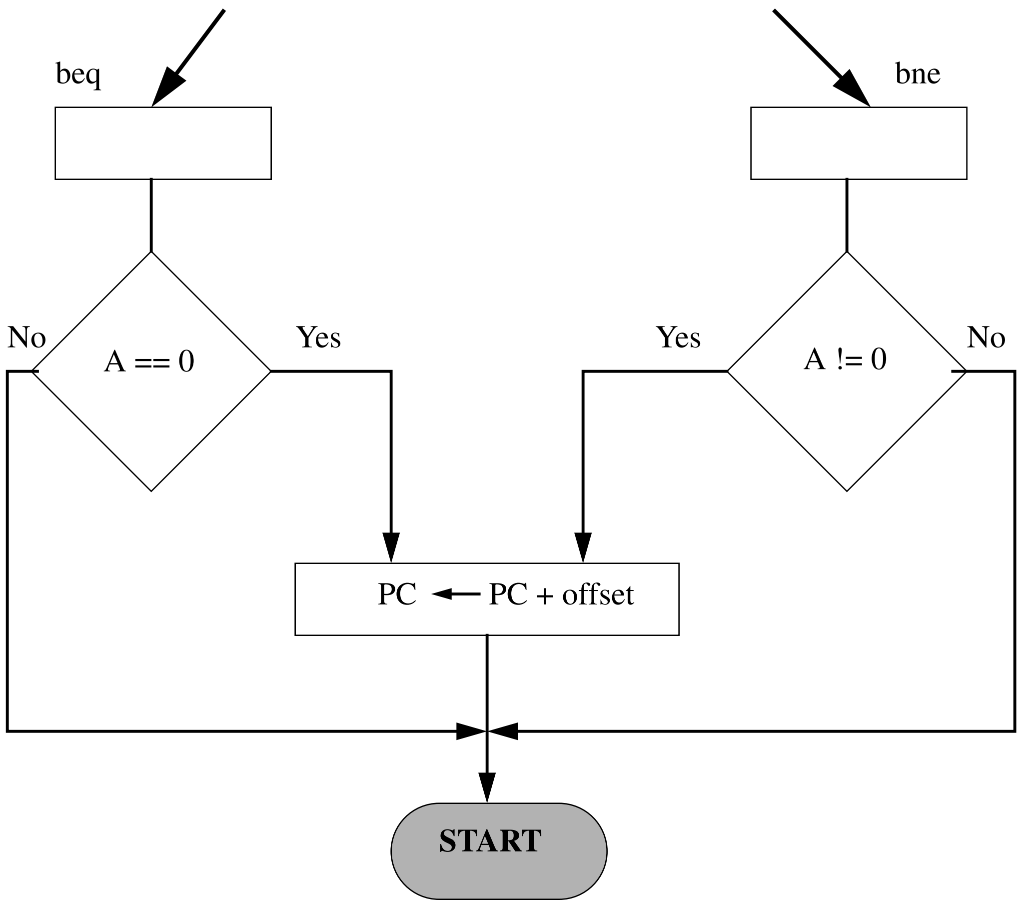

FIGURE 5.4 States in the state-diagram corresponding to the execution of branches.

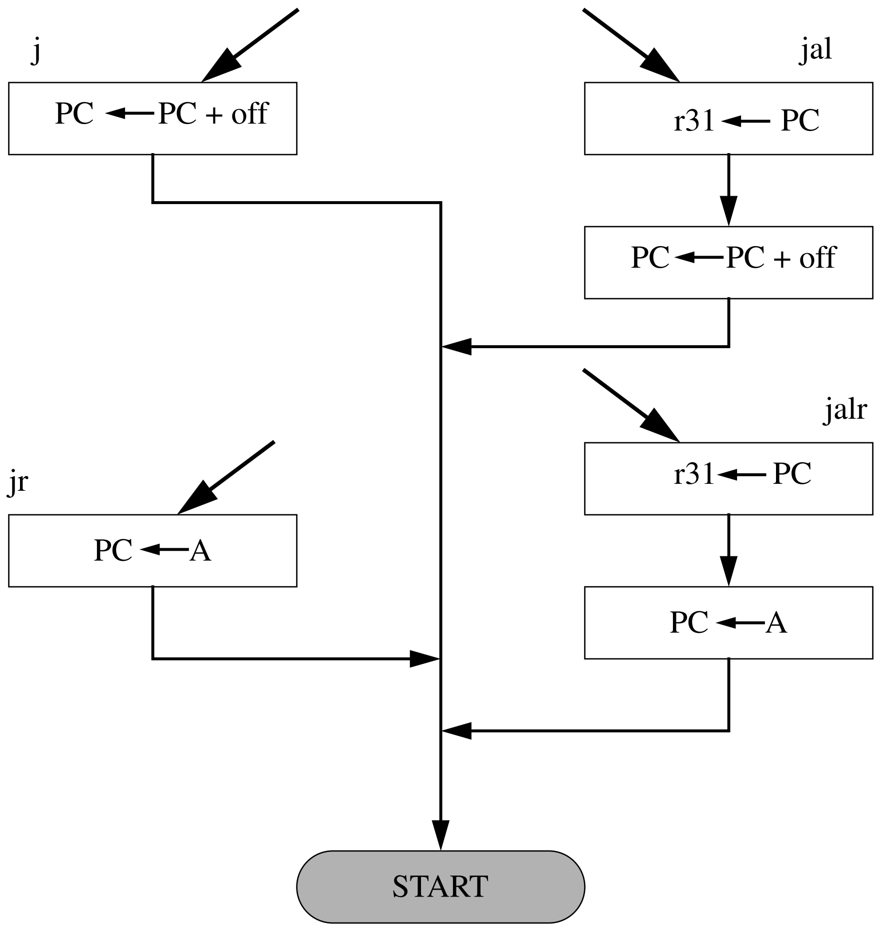

FIGURE 5.5 States in the state-diagram corresponding to execution of jumps. There seem to be six states at first look; however their number can be reduced to four by merging some of them.

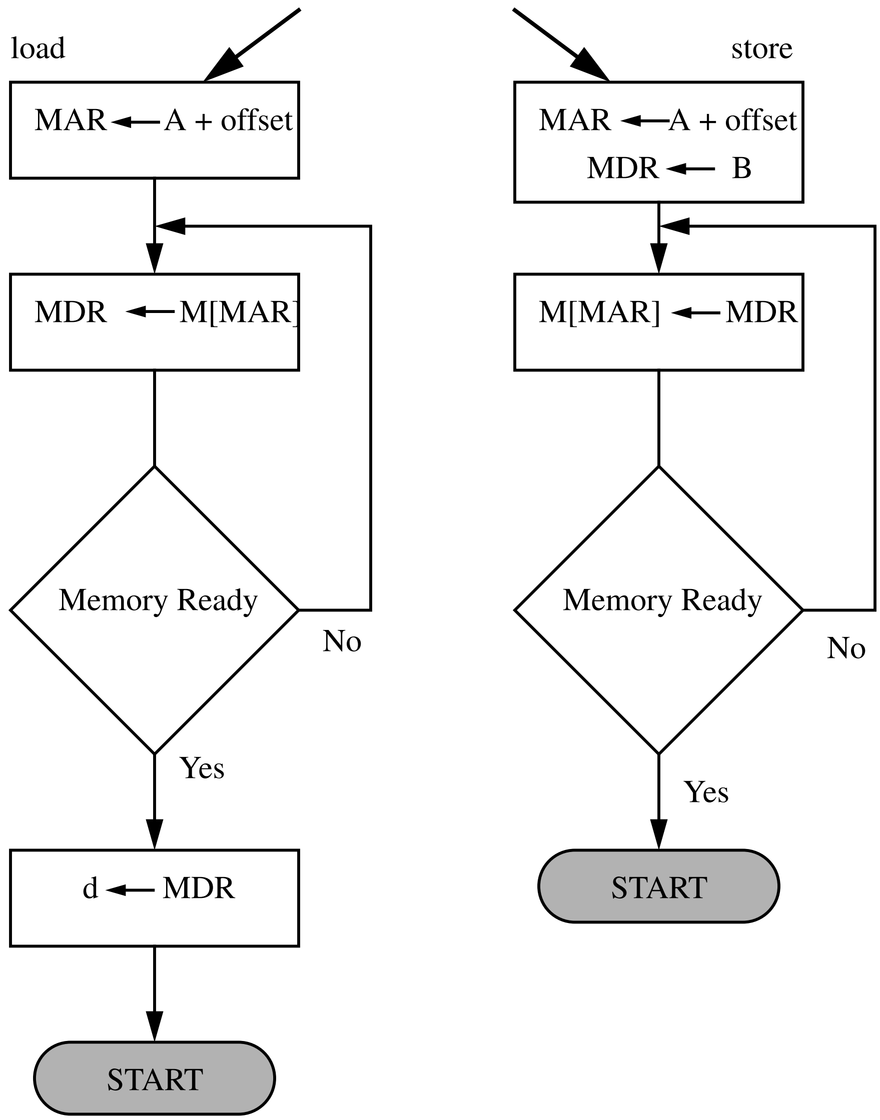

FIGURE 5.6 The states in the state-diagram corresponding to loads and stores.

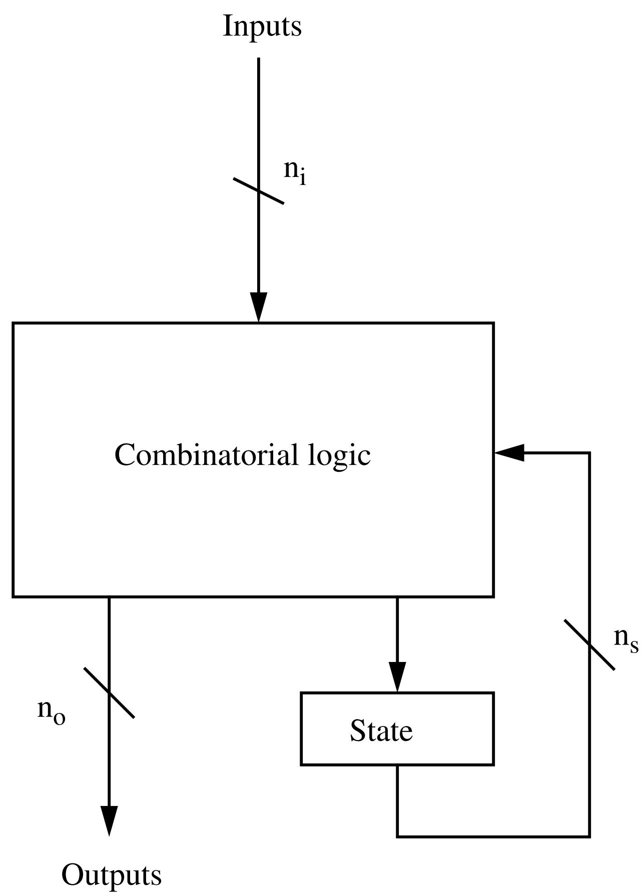

FIGURE 5.7 A general view of a Finite State Machine. Combinatorial logic and hardwired are synonyms.

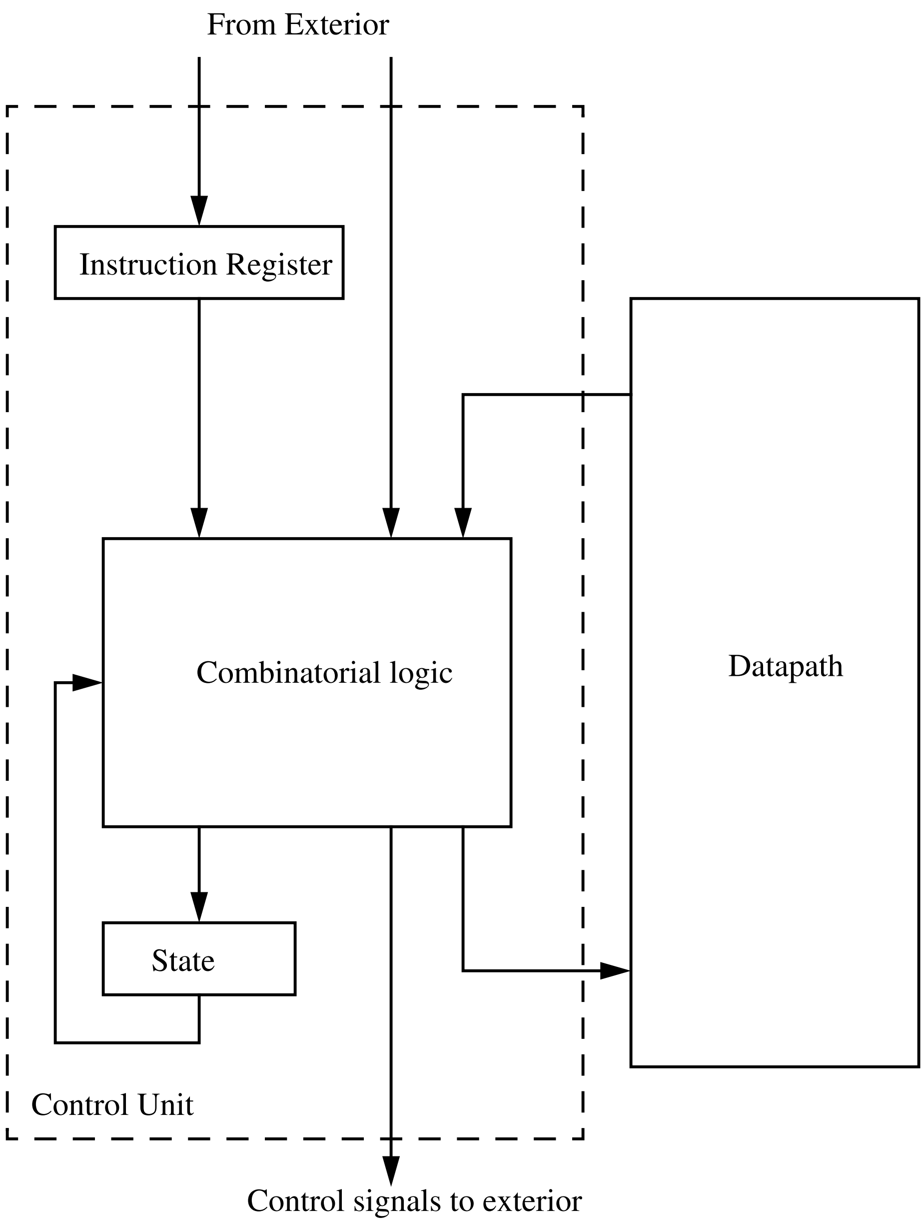

FIGURE 5.8 A Finite State Machine used as Control Unit. Inputs are coming from the Instruction Register (basically, the opcode), from the Datapath (conditions to be tested), and from the outside world (like Memory Ready, interrupts etc.).

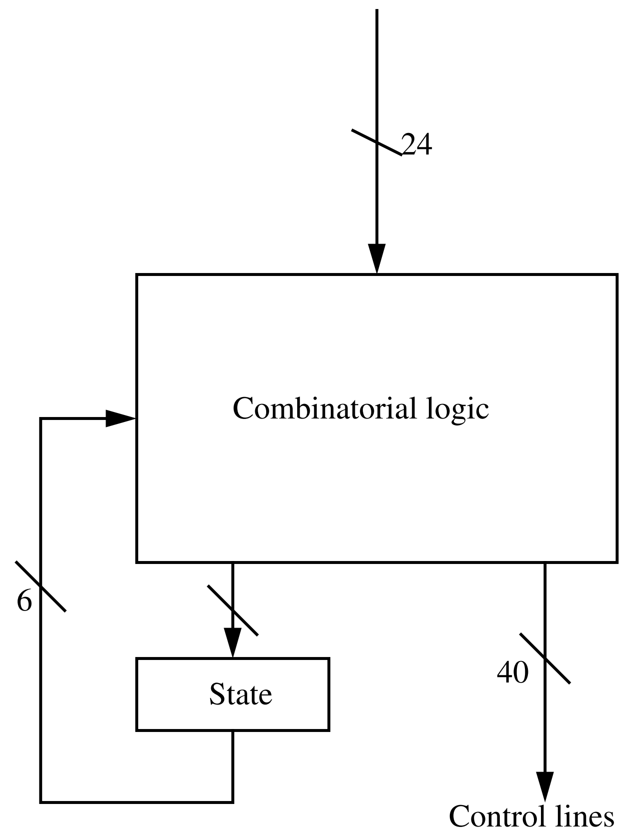

FIGURE 5.9 The Finite State Machine with numbers indicating how many inputs go to the Combinatorial circuit and how many outputs are there.

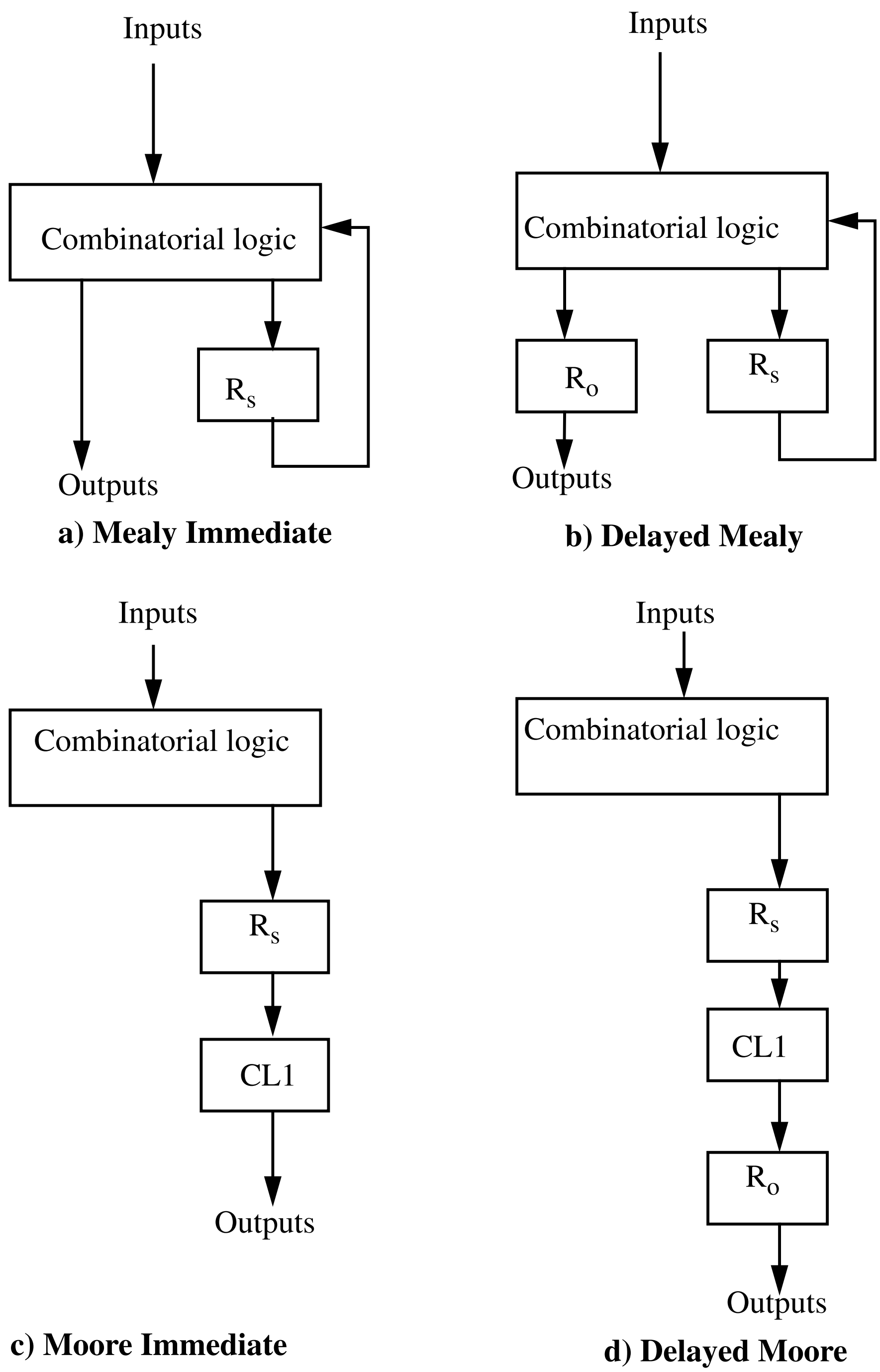

FIGURE 5.10 A classification of Finite State Machines. R_s is the state register. R_o is the output register, and CL1 is a combinatorial circuit which produces the outputs based on the state. Registers are clocked with the main clock signal (not represented). For Mealy machines the outputs are calculated based on the state and the inputs, while for Moore machines the outputs depend only on the state.

3. PLA implementation

A PLA organization resembles very much to that of a ROM with the big difference that only a handful of product terms (not necessarily minterms) are computed, those that are “added” to yield the output. Contrary to ROM implementation, the PLA implementation requires minimization, more precisely the system of function has to be minimized, in the sense that as many as possible common terms should be found between functions. PLA implementation is used in most Control Units in the today's CPUs.

Here are some of the problems related to this kind of implementation:

- **state-assignment:** changing the binary codes we assign to states may seriously affect the size of the PLA; not only is important what the binary codes are, but also the ordering in the state-diagram. Ordering of binary codes assigned to states is important also if we are concerned with the problem of combinatorial hazard (and we must be concerned as long as many of the control signals must be hazard-free). In this context note that the schematic diagram of a FSM in Figure 5.7 describes a particular kind of machine, one in which output directly come from the Combinatorial circuit. Figure 5.10 presents a more systematic classification of Finite State Machines. The basic difference between Mealy and Moore machines is that in a Mealy FSM the outputs are depend both of the state and of the inputs of CL in that state, while in a Moore machine the outputs depend only upon the state. Immediate machines yield output directly from the combinatorial logic and, as a consequence, output may have hazard. On the other hand delayed machine have clean, hazard-free outputs.

- **input signals signification:** binary input configurations affect the PLA in the same way as state binary configurations do. In this context the allocation of opcodes may significantly affect the design. Even if all possible binary combinations are used as opcodes, the order in which they are assigned to various operations is important.

5.5 Performance for Hardwired Control

This section presents a brief overview of the performance of hardwired control for our instruction set. Any instruction take at least 3 clock cycles only for fetch and decode, this in the case there is no waiting time for a memory cycle to complete. Then the ALU instructions require one more cycle to complete (that means that the ideal CPI for an add is 4), while loads require three more cycles (the ideal CPI for a load is 6). Recall the ideal CPI is the CPI with a zero delay memory subsystem. The real CPI takes into consideration the finite response time of the memory, and as such is always larger than the ideal one.

We must always keep in mind that the ultimate goal of the designer is to get the best CPU performance, i.e. to minimize the CPUtime which is expressed as:

$CPU_{time} = IC * CPI * T_{ck}$

At the CPU design level the parameters that can be the easiest affected are the CPI, through the sequence of states that embody the execution of an instruction, and Tck, both through technological refinements and organization inventions. Remember that for the store instructions we suggested a possible organization change in that a direct path from the B outputs of the register file to the MDR could save a state (that is a clock cycle in execution); Figure 5.7 reflects this option.

Example 5.2 CPU PERFORMANCE:

Stores represent a fraction f=20% of the instructions in a program. A direct path from the B outputs to the MDR reduces the store CPI from 6 to 5. By how much does the CPU performance improve? Suppose the original CPU is 3.5.

Answer:

CPUtime old = IC * CPIold * Tck
CPUtime new = IC * CPInew * Tck
CPInew = CPIold - f * 1   one clock cycle saved at every store
CPInew = 3.5 - 0.2 = 3.3

$$\frac{CPU_{time\ old}}{CPU_{time\ new}} = \frac{3.5}{3.3} = 1.06$$

which means an improvement by 6% in performance.
Note: we assumed in this example that the change in organization does not affect the clock rate.

It sounds simple but it is not that simple in practice. A direct connection between those two parts of the CPU means routing 32 wires in a structure where the space is not so easily available. Moreover such a change might affect the clock rate thus offsetting the gain in speed due to a reduced CPI.

Example 5.3 CPU PERFORMANCE:

Suppose you consider the same change in organization as in the previous example with the difference that this slows down the clock rate by 10%. What is the performance improvement?

Answer:

CPUtime old = IC * CPIold * Tck old
CPUtime new = IC * CPInew * Tck new
CPInew = CPIold -f*1one clock cycle saved at every store
CPInew = 3.5 - 0.2 = 3.3
Tck new = Tck old + 10%*Tck old = Tck old*(1+0.1)

$$\frac{CPU_{time\ old}}{CPU_{time\ new}} = \frac{3.5 * T_{ck\ old}}{3.5 * 1.1 * T_{ck\ old}} = 0.96$$

which means that the performance decreases in this case.

Improvements in the organization may decrease the CPI for some of the instructions or for all (this is a happy case). Figure 5.2 assumes that a fetch cycle starts with moving the PC into MAR. What if we remove this step? This means some connections from the PC to the address lines emerging from the CPU, plus some extra hardware (a multiplexor to select from MAR, in the case of load/store, plus a new control line that commands this MUX). We get for this price a reduction with one of the CPI for any instruction. Probably this is an improvement worth to be considered.

Exercises

5.1 How many compare operations are needed as a minimum? In the course we listed all possibilities.

5.2 Our machine is similar with many RISC machines at least in respect of basic concepts of Instructions Set design and hardware organization. These machines have one register (usually r0) tied to zero. Explain why. Do you see any need for such a decision?

5.3 Design a four register, multiple port register-file. It should contain two read ports and a write port.

5.4 Compute the CPI for every instruction in the Instruction set we defined and then the overall CPI considering the following frequencies of operations:

- ALU 50%
- set instructions represent 10% of ALU operations
- Jumps/Branches20%
  - simple jump represent 10%
  - jump and link are 10%
  - branches represent 80%
- Load/Store25%
  - loads represent 50%
  - stores represent 50%

Assume that branches are taken with the same frequency as they are not taken.

5.5 Draw in detail the sequence of tests that follows the Q2 state in Figure 5.2. For this you have to choose opcodes for instructions and codes in the func field for different arithmetic operations (as you can observe the func field is not absolutely necessary, our opcode field can accommodate 64 different instructions). Is there some way to modify the opcodes such that the sequence of tests gets simpler?