Branching

The MIPS programs we have seen to this point involve a sequence of instructions that are unconditionally executed. In other words every instruction is executed and all instructions are executed in the order that they appear in the program. However, high level languages employ programming constructs such as while-loops and if-then-else conditionals where blocks of code are conditionally executed. This behavior must be preserved when these blocks of code are compiled to assembly language instruction sequences. Such conditional execution behavior is achieved with the help of a few additional instruction types - *branch* instructions. These instructions effectively alter the flow of program control from the otherwise strict in-order execution This chapter is focused on the description and use of branch instructions.

Conditional Branch Instructions

Consider the SPIM code example shown in Figure 1. Imagine an array of 8 values stored in memory and a code sequence that must compute the sum of every other value. Register $t0 is used to keep track of the offset from the starting address of the array which is initially 0. Register $t1 is used to store the sum. The program includes one block of three instructions that is repeated four times. These instructions i) add 8 to the offset to compute the address of the next element, ii) load the next element into a register, and iii) add this element to the sum being accumulated in $t1.

```
            .data
L1:         .word 2,4,6,8,10,12,14,16 # the array

            .text
            add $t0, $0, $0          # initialize $t0=0
            add $t1, $0, $0          # initialize the sum = 0
            lw $t2, L1($t0)          # load first value
            add $t1, $t1, $t2        # add to sum
            addi $t0, $t0, 8         # computer addr of two elements down
            lw $t2, L1($t0)          # load second value
            add $t1, $t1, $t2        # add to sum
            addi $t0, $t0, 8         # computer addr of two elements down
            lw $t2, L1($t0)          # load third value
            add $t1, $t1, $t2        # add to sum
            addi $t0, $t0, 8         # computer addr of two elements down
            lw $t2, L1($t0)          # load fourth value
            add $t1, $t1, $t2        # add to sum
            li $v0, 10
            syscall                  #end the program
```

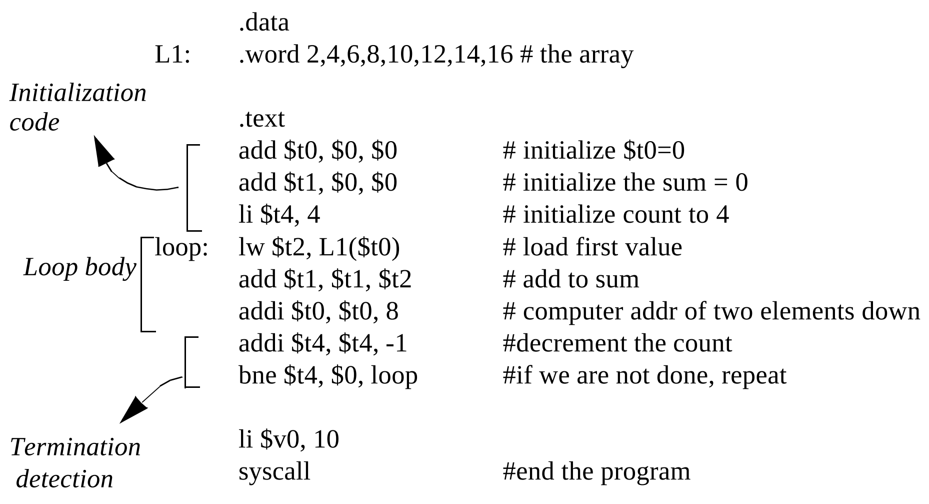
FIGURE 1. **Example SPIM program**

```
addi $t0, $t0, 8
lw $t2, L1($t0)
add $t1, $t1, $t2
```

This group of three instructions is repeated four times. Why not just update the value of $t0 and execute the same three instructions four times, or in other words, write a *while-loop*? To do so we need some way to alter the flow of program control so that the next instruction that is executed is not necessarily the instruction following the current instruction, but rather some other instruction. The most basic form of an instruction for altering the flow of control in a program is the branch instruction. For example, consider the following instruction.

```
bne $t0, $t1, loop
```

The instruction is read as "branch on not equal". If the contents $t0 is not equal to the contents of $t1 then the next instruction that is executed is the instruction labeled loop. If the contents of the two registers are equal, then the instruction execution proceeds as usual and the following instruction (in program order) is fetched and executed. Now with such as instruction we can rewrite the



```
                .data
        L1:     .word 2,4,6,8,10,12,14,16 # the array

                .text
                add $t0, $0, $0       # initialize $t0=0
                add $t1, $0, $0       # initialize the sum = 0
                li $t4, 4             # initialize count to 4
        loop:   lw $t2, L1($t0)       # load first value
                add $t1, $t1, $t2     # add to sum
                addi $t0, $t0, 8      # computer addr of two elements down
                addi $t4, $t4, -1     #decrement the count
                bne $t4, $0, loop     #if we are not done, repeat

                li $v0, 10
                syscall               #end the program
```

FIGURE 2. The SPIM program in Figure 1 rewritten as a while loop

program in Figure 1 to be of the form shown in Figure 2. In this case we create a *while-loop*. The loop is comprised of three basic code components. The first block of code is the initialization code. Counters, offsets, and other relevant values are initialized. For example in this case a counter is initialized with the value of 4 to keep track of the number of times this group of instructions is executed. The second code component is the loop body. This is the block of code that is executed repeatedly. In Figure 2 it is the three instructions that fetches the value, adds the value to the sum and updates the offset to the next value. The final code component is the termination detection: has the loop executed the correct number of iterations? Each time through the loop the counter is decremented and the result is tested for equality with the value 0 use the bne instruction. If the counter, which is maintained in $t4, has a non-zero value then more work needs to be done. The next instruction executed is the instruction at the beginning of the loop body and the program is said to branch back to the instruction labeled loop. The instruction labeled loop is referred to as the *branch target.* When the while-loop has executed four times and the count has dropped to 0 the execution of the branch instruction will cause program execution to continue with the next instruction rather than branching back to the branch target and the program terminates. Such instructions are referred to as conditional branches since the choice of the next instruction to be executed is based on the outcome of a test of a condition, in this case a condition involving the equality of two registers.

It is important to understand that the programs of Figure 1 and Figure 2 are functionally identical. It was possible to rewrite the program in Figure 1 into the shorter and more compact form shown in Figure 2 because of the availability of the bne instruction. The SPIM ISA also includes a beq instruction which works similarly: in this case the contents of the two registers must be equal for the program to branch.

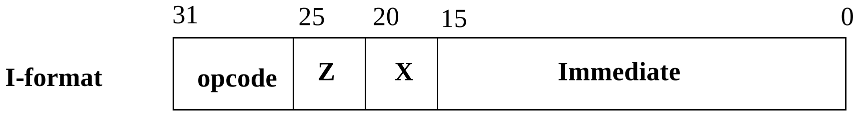


FIGURE 3. **I-format**

As an exercise rewrite the program in Figure 2 using a beq instruction rather than the bne instruction [*Hint*: Place the value 4 in a register and count up to 4 rather than down to 0]. You can test this program with the simulator.

How are branch instructions encoded? All branch instructions use the I-format shown in Figure 3. Note the presence of the offset. The branch offset is computed as the number of *words* between the branch instruction and the branch target. The offset is computed such that if it is added to the PC the result will be the address of the branch target instruction.

How does the hardware implementation execute a branch instruction such as a beq $t0, $t1, loop instruction? First the contents of $t1 are subtracted from the contents of $t0. If the result is 0 then the branch is taken and the PC must be updated to contain the address of the branch target. In the MIPS implementation the offset encoded with the branch instruction is expressed in words rather than bytes. The hardware implementation of the branch instruction will multiply this offset by 4 to generate the value of the offset in bytes. This byte offset will be added to the current contents of the PC to produce the branch target address.The PC will now be loaded with the branch target address and the instruction fetch will continue from this new address. A 16-bit word offset corresponds to an 18-bit byte offset (multiply by 4 which is equivalently a left shift by two bit positions). Offsets may be positive or negative numbers with twos complement representation for the negative numbers. Thus the range of byte addresses that can be produced for a branch target is given by the following.

$$PC - (2^{17}) \le TargetAddress \le PC + (2^{17} - 1)$$

Example: Consider code block shown in Figure 2 and the encoding of the branch instruction. The opcode for the bne instruction is 5. When the branch instruction is being executed the contents of the PC is the address of the encoded branch instruction. If the branch is taken the next instruction to be fetched would be the instruction labeled loop. What is the relative difference between the address of the branch instruction and the address of the branch target? In other words what is the value, in words, between the addresses of the branch instruction and the target. In this case it is -4 since the branch target address is less than the branch instruction address. The branch instruction can now be encoded with the value of -4 in the offset field to produce the 32-bit encoded value of 0x1520fffc. Note that the offset is represented in two's complement form. When the branch is executed and if the branch is taken then the offset is multiplied by 4 and added to the contents of the PC to produced the new value of the PC. Now we see why the offset is in two' complement form - so as to effect the subtraction when the program branches "back" to an instruction at a smaller memory address.

In summary the following two native conditional branch instructions were introduced in this section and are described in Table 1.

Instruction	Opcode	Operation
beq $t0, $t1, loop	000100	If the contents of $t0 is equal to the contents of $t1, next instruction that is fetched is the instruction labeled loop
beq $t0, $t1, loop	000101	If the contents of $t0 is not equal to the contents of $t1, the next instruction that is fetched is the instruction labeled loop

TABLE 1. The conditional branch instructions

Unconditional Branch Instructions

It is also quite useful to have an instruction which unconditionally branches to a labeled instruction. For example, consider the case where it is necessary to conditionally execute a block of code as shown in the pseudocode of Figure 4. Register $t1 will have the sum or difference of $t2 and $t3 depending on the value in $t0. Only one block of code (corresponding to the sum or difference) must be executed. The assembly language implementation is also shown in Figure 4. Since instructions are physically sequential the code in the *then* part of the code must precede the instructions corresponding to the *else* part of the code (or vice versa, the problem is the same). Only one of these code blocks must be executed. In Figure 4 the jump instruction is introduced.

```
j exit
```

The behavior of the jump instruction is such that the next instruction executed is the instruction labeled exit. This transfer of control is unconditional and always takes place. It is also the behavior we desire here since if the *then* part of the code is executed as a result of the test of the value in $t0, we would

Pseudocode

```
if ($t0 != 0) then -- check if $t0 not equal to 0
        $t1 = $t2 + $t3
else
        $t1 = $t2 - $t3
end if;
```

Assembly Code

```
        bne $t0, $0, else        # check if the contents of $t0 = 0
                                 # Branch if not equal
        add $t1, $t2, $t3        # this is the then part of the code
         j exit                  # now we need to jump over the
                                 # else part of the code
else:   sub $t1, $t2, $t3        # this is the else part of the code
exit:   move $t4, $t1            # the program continues using
                                 # the value in $t1
```

FIGURE 4. An example of the implementation of the if-then-else construct

always require that the execution of the *else* part of the code be avoided. The transfer of control to the first instruction following the *else* part of the code achieves this behavior. The instruction labeled exit is referred to as the *jump target.*

How are jump instructions encoded? The jump instruction provides the absolute address of the destination instruction and is encoded using a new instruction format: the J-format shown in Figure 5 which utilizes a 6-bit opcode as in the R-format and I-format instructions. However, since 6 bits of the instruction encoding is necessary for the opcode only 26 bits remains to specify the address of the jump target. Thus addressing is limited to 2^{26} or 64 Mbytes of instruction memory which is a rather small address space by modern standards. Therefore the MIPs implementation utilizes a clever scheme. The 26 bit address encoded in the instruction is the word address. It can be converted to a byte address by multiplying by 4. Multiplication by 4 can be effected by a left shift of 2 bits. The result is a 28 bit byte address or a 256 Mbyte address range. What about the remaining 4 bits? In our current implementation we assume that upper four bits of the PC remain unchanged when a jump instruction is implemented. Thus the 28 bit byte address produced from the jump instruction replaces the lower 28 bits of the PC. As long as programs fit within the 256 Mbyte

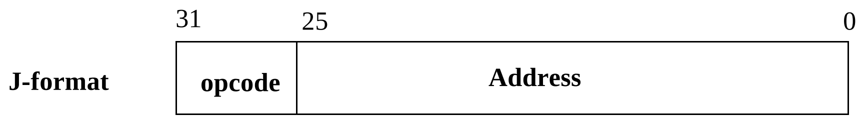


FIGURE 5. **J instruction format**

address space this solution is correct. When the address spaces are larger then other techniques must be applied. Let us look at an example of how these instructions are encoded.

Example: The target label, proc, has a value of 0x00400040. What would be the hexadecimal encoding of the j proc instruction? The first observation is that the value of label is a byte address. The jump instruction encodes a 26-bit word address. How do we obtain the corresponding word address? First a few observations. Word addresses start at every fourth byte. For example, consider the memory address 0x8. This location corresponds to the starting address of the second word in memory. Similarly the byte location 0xc corresponds to the starting byte of the third word in memory. Thus we see in general the word address is obtained from the byte address by dividing the byte address by 4! Division is performed by a right shift of two bits. The opcode for the jump instruction is 2. Now we can compute the encoded value of this instruction which is 0x08100010

Finally, what if a 26-bit word address does not suffice and we really do want to generate and use a 32-bit byte address? The MIPS instruction set provides an additional instruction for this purpose.

```
jr $t0
```

This instruction will cause the contents of register $t0 to be placed in the PC. Since registers are 32 bits this instruction provides programs with access to the full 32-bit addresses. Either the programmer will have to explicitly construct the address in a register or the compiler/assembler will have to construct these addresses as necessary. When we review the implementation of functions and procedure calls we will see a very common use of this instruction.

In the summary, the two instructions introduced in this section are shown in Table 2.

Instruction	Opcode	Operation
j func	000010	The next instruction that is fetched is the instruction labeled func
jr $t0	000011	The next instruction fetched is at the 32-bit address stored in $t0

TABLE 2. The unconditional branch instructions

General Branch Conditions

While beq and bne are useful instructions we are often more interested in more general conditions such as branch-greater-than-or-equal to or branch-less-than or branch-greater-than. From a hardware perspective supporting each such branch instruction in hardware is not a cost-effective or performance-effective solution. Such general branch conditions are really more of a programming convenience for assembly language writers and compiler code generators. What would be desirable is a small number of native instructions that are actually implemented in hardware and from which more complex branch conditions can be constructed. Let us assume that the beq and bne instructions are implemented in the MIPs hardware. To this pair of instructions is added the slt instruction which is read as "set less than". Using these three instructions we can synthesize all of the general branch conditions. The slt instruction operates as follows.

```
slt $t0, $t1, $t2
```

The behavior of this instruction is as follows. If the contents of $t1 is less than the contents of $t2 the contents of $t0 is set to 1. Otherwise the contents of $t0 is set to 0. The behavior of the instruction itself is simple enough. The power comes from combining it with the beq or bne instructions. Consider the following implementations of various branch conditions as captured in Table 3.

Branch Instruction	Meaning	Implementation
blt $t0, $t1, loop	Branch to loop if $t0<$t1	slt $t2, $t0, $t1 bne $t2, $0, loop
bge $t0, $t1, loop	Branch to loop if ($t0>$t1) or ($t0=$t1). This is equivalent to testing for $t1<$t0.	slt $t2, $t1, $t0 bne $t2, $0, loop

TABLE 3. Examples of synthesizing general branch conditions

Each of the general branch instructions in the first column of Table 3 can be realized with the code sequence shown in the last column. In practice the MIPS instruction set includes instructions such as those shown in the first column. The MIPS assembler will translate these branch instructions into equivalent instructions sequences comprised of native instructions as shown in the last column. The branch instructions shown in the first column are referred to as *pseudoinstructions* since they are not actually implemented in hardware but rather are translated by the assembler into those instructions that are implemented in hardware. In general you will find other pseudoinstructions that are not branch instructions. For a complete list of the branch instructions supported by in SPIM refer to the SPIM documentation.

We have seen that while beq, bne, slt, and slti (one of the operands can be an immediate operand) are implemented by processors implementing the MIPs instruction set more complex branch conditions are replaced by the corresponding simpler instruction sequences by the assembler. In this manner the programming interface is simplified by the presence of powerful instructions while the hardware is simplified by reducing the number of instructions that are actually implemented in hardware.

In summary the instructions introduced here are shown inTable 4

Instruction	Opcode	Operation
slt \$t0, \$t1, \$t2	101010	If the contents \$t1 is less that the contents of \$t2, the contents of \$t0 is set to 1.
slti \$t0, \$t1, 12	001010	The same as slt except that one of the operands may be a immediate.

TABLE 4. Special instructions to support general branches

Modifications to the Datapath

The single cycle datapath must be modified to support the beq, bne, jmp, and slt instructions. This section only describes the additions necessary to support the beq and jmp instructions. Solutions for the remaining and other instructions follow a similar approach.

Two parts of the datapath are affected: the instruction fetch logic and the controller. Let us first consider the instruction fetch logic. During normal execution the next instruction to be executed is available in memory at the location following the current instruction. However, with the addition of the beq and jmp instructions the next instruction may be fetched from one of three possible sources: a branch target, a jump target, or PC+4. Both target addresses must be computed and the correct one selected. These target addresses can be computed as part of the next instruction logic as illustrated in Figure 6.

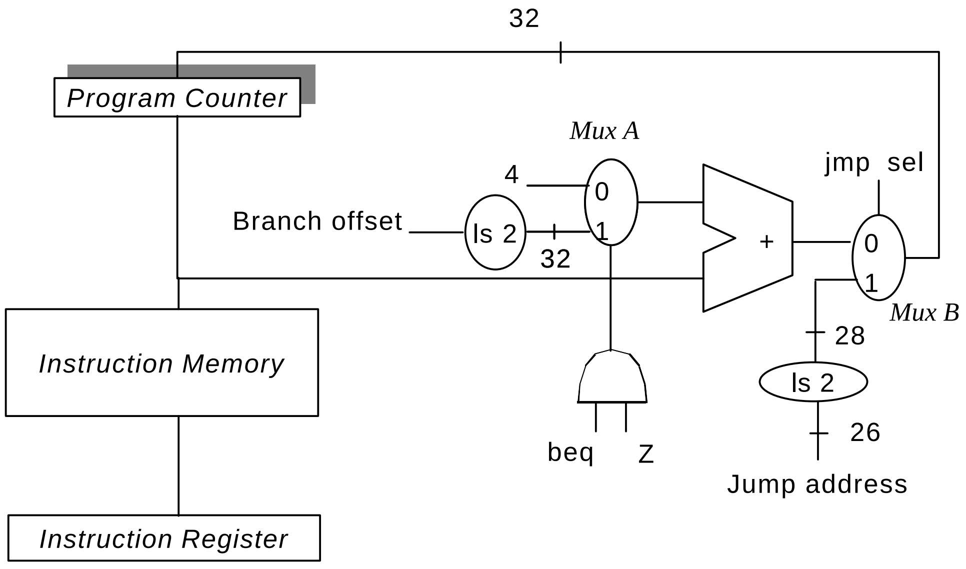


FIGURE 6. Next instruction fetch logic

The branch offset is obtained from the output of the sign extension unit. The jump address is available from the lower 26 bits of the IR. Both values are expressed in words and therefore must be multiplied by 4 to express the equivalent value in bytes. Multiplication by 4 can be realized by an arithmetic left shift by 2 bits. The first multiplexor, Mux A, must select the value to be added to the PC. The choice is between a branch offset (now expressed in bytes) and the value 4. The former is chosen if this instruction is a branch instruction (beq is true) and the zero bit from the adder subtractor (Z) is also true denoting the contents of the two registers identified by beq instruction are indeed equal. If either of these conditions is false then the next instruction to be executed is the following instruction whose address is given by PC+4. We see from the figure that the control signal for Mux A is set accordingly.

The second multiplexor, Mux B, selects the jump address if this is a jump instruction (jmp sel is true). Otherwise the output of the first multiplexor is retained as the address of the next instruction. One point to be noted here is that the jump address is 28 bits. The PC is 32 bits. What is not shown in Figure 6 is that the higher order 4 bits of the PC are concatenated with the 28 bits of the jump address to produce the full 32-bit value of the jump address.

Since the beq instruction requires a 16-bit offset the instruction is encoded using the I-format as shown in Figure 3. Note the following problem. The implementation of the branch instruction uses the adder/subtractor unit to determine if the values in the two registers are equal by subtracting the contents of the two registers. Recall that the adder/subtractor produces a Z status bit that denotes whether the result of the most recent operation was 0. This Z status bit is used in the next instruction logic. The I-format identifies one source register and a destination register. Thus both registers specified in the instruction are source registers. However the I-format identifies one source register and a destination register! This issue can be addressed in the following manner. The input of the Y address port of the register file must be able to choose between the Z register address field and Y register address field of the IR. A single-bit control signal, RegSrc will be used select the source register whose contents is to be placed on the Y-bus. For all register-to-register and lw/sw operations this will the Y register address field from the IR. For the beq instruction this will be the Z register address field. The resulting register file with the modified addressing logic is shown in Figure 7. The new control signal, RegSrc, is used to control the source of the address of the register whose contents are placed on the Y bus.

Finally what about the impact on the controller? The beq and jmp instructions are assigned opcodes and added to the truth table that describes the relationship between instruction opcodes and control signal values. The controller truth table now appears as shown in Table 5. Two additional control signals must be added signifying whether an instruction is a beq or jmp instruction. These control signals are used in the instruction fetch logic shown in Figure 6. The controller can now be synthesized into a digital logic implementation.

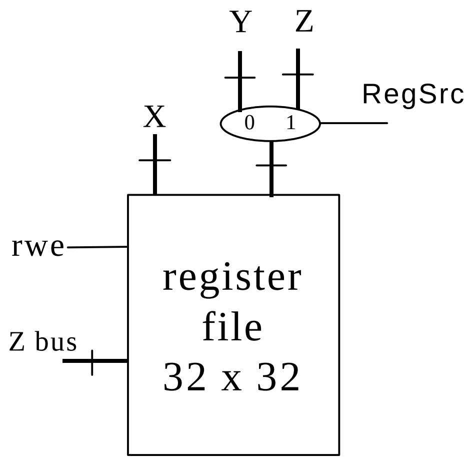


FIGURE 7. Modified register file addressing logic

Opcode (Instr)	rwe	imm en	au en	$\bar{a}$/s	lu en	lf	su en	st	st en	ld en	r/$\bar{w}$	msel	RegSrc	beq	jmp
100000 (add)	1	0	1	0	0	0000	0	00	0	0	0	0	0	0	0
100010 (sub)	1	0	1	1	0	0000	0	00	0	0	0	0	0	0	0
100100 (and)	1	0	0	0	1	1000	0	00	0	0	0	0	0	0	0
100101 (or)	1	0	0	0	1	1110	0	00	0	0	0	0	0	0	0
100011 (lw)	1	0	0	0	0	0000	0	00	0	1	1	1	0	0	0
101011 (sw)	0	0	0	0	0	0000	0	00	1	0	0	1	0	0	0
001000 (addi)	1	1	1	0	0	0000	0	00	0	0	0	0	0	0	0
000000 (nop)	0	0	0	0	0	0000	0	00	0	0	0	0	0	0	0
000110(beq)	0	0	1	1	0	0000	0	00	0	0	0	0	1	1	0
000010(jmp)	0	0	0	0	0	0000	0	00	0	0	0	0	0	0	1

TABLE 5. Controller truth table

Summary of Addressing Modes

This chapter has introduced a few new instructions and in the process a few new addressing modes. Recall that an addressing mode reflects how objects are addressed. The addressing modes we have seen to date are summarized below.

Register addressing: The operands are identified by the register name, for example, as in $t0, or $t1.

Immediate addressing: The operand is specified and encoded with the instruction. For example, as in addi $t0, $t1, 4. The value 4 is an immediate operand.

Base or Index addressing: Adding an offset to a base address as required by the lw and sw instructions is an example of this mode for constructing memory addresses.

PC-Relative addressing: Computing an address by adding an offset to the PC computes an address relative to the PC and hence the name for this addressing mode. Branches utilize PC-Relative addressing.

Register Indirect addressing: Using the contents of a register as an address necessitates first addressing a register to obtain an address. Therefore this is regarded as indirectly obtaining an address - one addressing mode is necessary to obtain a valid memory address.

The preceding are the most common addressing modes and will be found in some form in most any modern instruction set.