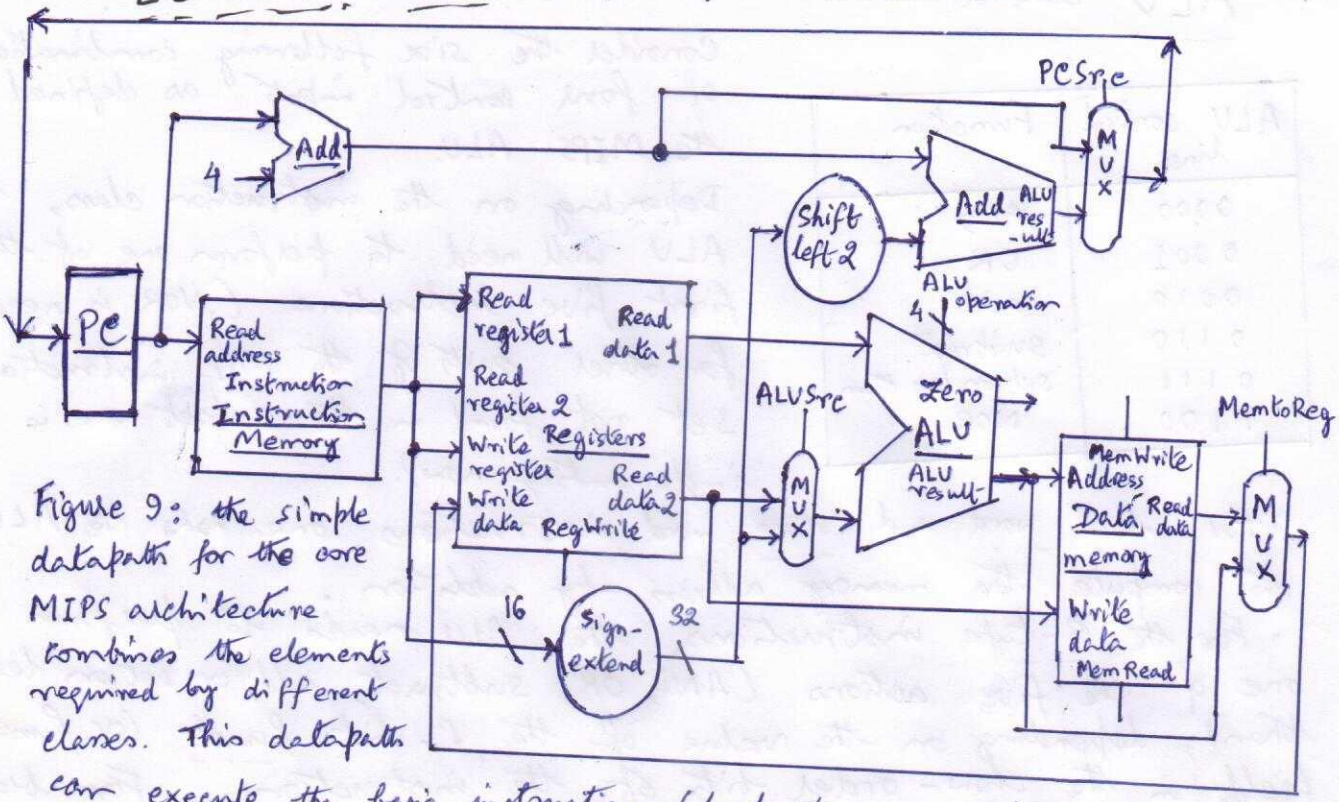


Figure 9: the simple datapath for the core MIPS architecture combines the elements required by different classes. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. One additional multiplexor only is needed to integrate branches.

- On completing the simple datapath, one can now add the control unit. The control unit must take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control. As the ALU control is different in a number of ways, it will be useful to design it first before one designs the rest of the control unit.

A Simple Implementation Scheme

Let us look at the simplest possible implementation of the chosen MIPS subset. One can build this simple implementation using the datapath (that has been discussed above) and adding a simple control section (function). This simple implementation covers load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than (slt). One can later enhance the design to include a jump ~~section~~ instruction (j).

P.T.O.

ALU control

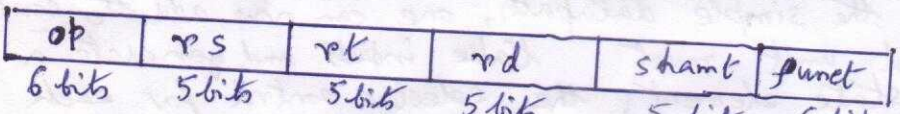
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Consider the six following combinations of four control inputs as defined by the MIPS ALU.

Depending on the instruction class, the ALU will need to perform one of these first five instructions (NOR is needed for other parts of the MIPS instruction set not found in the subset one is implementing now).

- For load word and store word instructions, one uses the ALU to compute the memory address by addition.
- For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction. For branch equal, the ALU must perform a subtraction.

[MIPS fields are given names to make them easier to discuss:

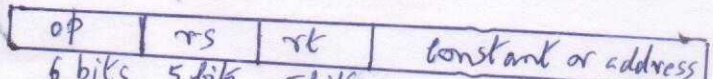


R-format or R-type instruction (R: register)

The meaning of each name of the fields in MIPS instructions is:

- op: basic operation of the instruction, traditionally called 'opcode'
- rs: the first register source operand
- rt: the second register source operand
- rd: the register destination operand; it gets result of operation.
- shamt: shift amount (used with 'shift' instructions)
- funct: (Function): this field (often called function code) selects the specific variant of the operation in the 'op' field

- problem occurs when an instruction needs longer fields than those shown above. For example, the load word inst^n must specify two registers and a constant. If the address were to use of one of the 5-bit fields in the format above, the constant within 'lw' inst^n will be limited to only 2^5 or 32. As 5-bit ~~field~~ field is too small to be useful, one uses I-type (for 'immediate') or I-format rather than above R-type (for 'register') or R-format.



16-bit address means a load word inst^n can load any word within a region of $\pm 2^{15}$ or 32,768 bytes (or $\pm 2^{13}$ or 8192 words as 1 word = 4 bytes) of the address in the base register rs.]

A Simple Implementation Scheme - ALU control (Contd.)

- One can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which one calls ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for beq, or determined by the operation encoded in the funct field (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations (showed previously).

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Figure 10: How the ALU control bits are fixed depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are in binary. Note that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; which is therefore shown as 'XXXXXX'. When ALUOp value is 10, the function code is used to set the ALU control input.

- Multiple levels of decoding - that is, main control unit generates the ALUOp bits, which are used as input to ALU control that generates the actual signals to control the ALU unit — is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially increase the speed of the control unit. Such optimizations are important, as the speed of the control unit is often critical to clock cycle time.

P.T.O.

- To implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the four ALU operation control bits, note that only a small number of the $2^6 = 64$ possible values of the function field are of interest, and the function field is used only when the ALUOp bits equal 10; so, one can use a small piece of logic that recognizes the subset of possible values and generates the ALU control bits

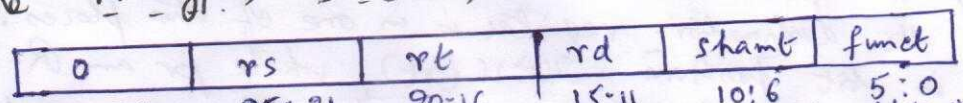
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Note: the full truth table (with $2^8 = 256$ entries) are very large, and one does not care about the value of ALU control for many of these input combinations

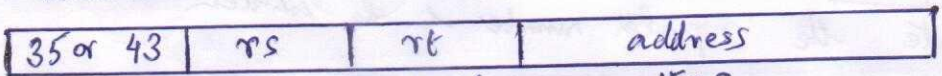
Figure 11: The Truth table for the 4 ALU control bits (called operation) Only the entries for which the ALU control is asserted are shown. Why are don't care entries added? For, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don't care terms and are replaced with XX in the truth table.

Designing the Main Control Unit

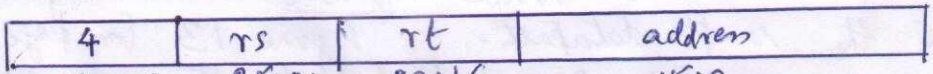
- As we have seen how to design an ALU that uses the function code and a 2-bit signal as its control inputs, we should now look at the rest of the control. As a starting point, let us identify the fields of an instruction and the control lines that are needed for the datapaths shown in Figure 9.
- To understand how to connect the fields of an instruction to the datapaths, one can review the formats of the three instruction classes: the R-type, branch, and load-store instructions.



a) R-type instruction



b) Load or store instruction



c) Branch instruction

Figure 12: the three instruction classes (R-type, load and store, and branch) use two different instruction formats. [N.B. jump instructions use another format, which is not shown here]

(a) Instruction format for R-format instructions, all of which have an opcode of '0', have three register operands: rs, rt and rd. The ALU function is in the funct field and is decoded by the ALU control design. The R-type instructions that we implement are add, sub, AND, OR, and slt. (The shamt field is for shifts, which are ignored now)

(b) Instruction format for load (opcode = 35_{10}) and store (opcode = 43_{10}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory.

(c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC+4 to compute the branch target address

P.T.O.

As observed from Fig. 12 (about the instruction format)
1) the op field, which is called the opcode, is always contained in bits 31:26. Let us refer to this fields as Op[5:0].
2) the two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
3) the base register for load and store instructions is always in bit positions 25:21 (rs).
4) the 16-bit offset for branch equal, load, and store is always in positions 15:0.
5) the destination register is in one of two places. For a load, it is in bit positions 20:16 (rt), while for an R-type instruction, it is in bit positions 15:11 (rd). Thus one needs to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

* Using the above information, one can add the instruction labels and extra multiplexor (for the Write register number input of the register file) to the simple datapath. Figure 13 (on page 3/10) shows these additions and the ALU control block, the write signals for the state (memory) elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.
• Figure 13 shows seven single-bit control lines and the 2-bit ALUOp control signal. As it has already been defined how the ALUOp control signal works, it is useful to define what the seven other control signals do informally before one determines how to set those control signals during instruction execution. Figure 14 describes the function of these seven control lines.
• Having looked at the function of each of the control signals, one can consider as how to set them. The control unit can set all except one of the control signals based solely on the opcode field of the instruction. The PCSrc control line is the exception. That control line should be asserted if the instruction is branch on equal (a decision that the control unit can make) and the Zero output of the ALU, which is used for equality comparison, is asserted. To generate the PCSrc signal, one will need to AND together a signal from the control unit, which one calls Branch, with the Zero signal out of the ALU.

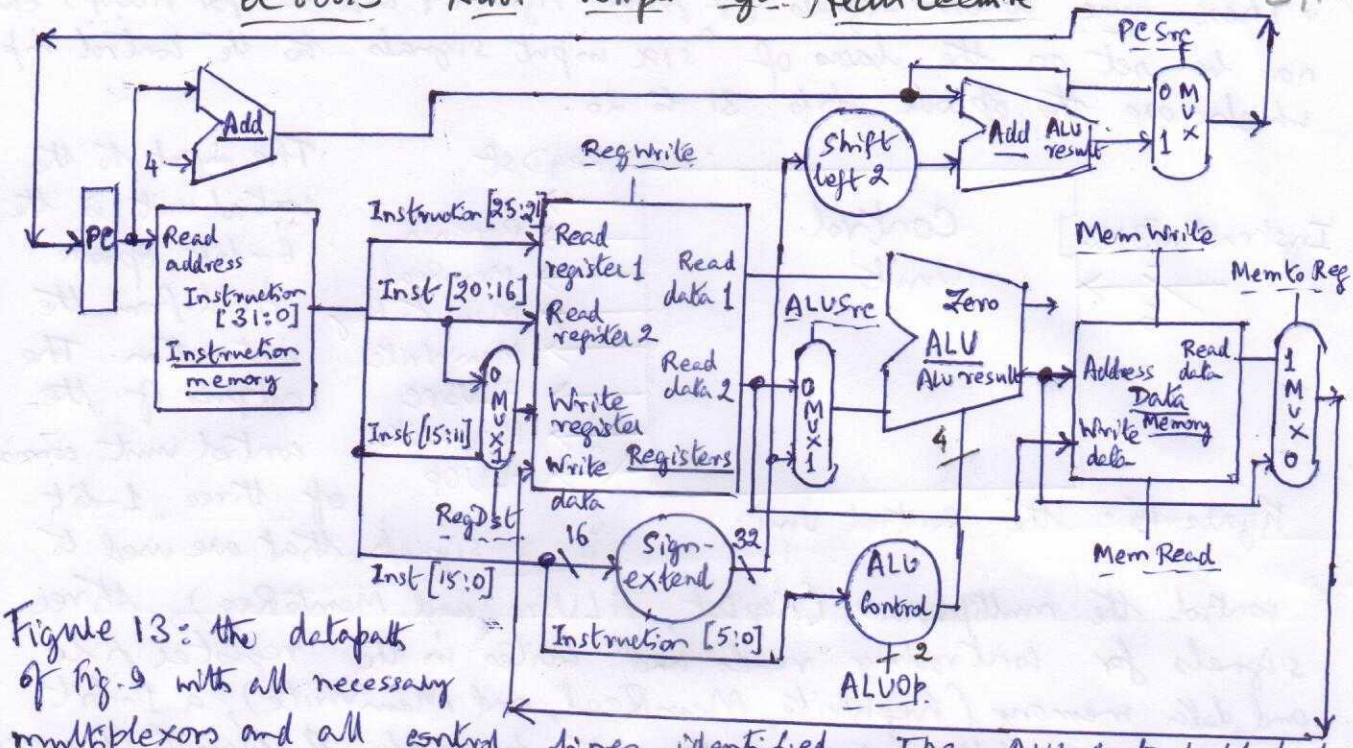


Figure 13: the datapath of Fig. 9 with all necessary multiplexors and all control lines identified. The ALU control block has been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	PC is replaced by the output of the adder that computes the value of PC+4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by address input are put on the Read data output
MemWrite	None.	Data memory contents designated by address input are replaced by the value on the Write data input
MemtoReg.	The value fed to the register Write data input comes from the ALU.	Value fed to the register Write data input comes from the data memory.

Figure 14: Effect of each of the seven control signals

P.T.O.

• These nine control signals (7 (seven) from Figure 14 and two for ALUOp) can now be set on the basis of six input signals to the control input, which are the opcode bits 31 to 26.

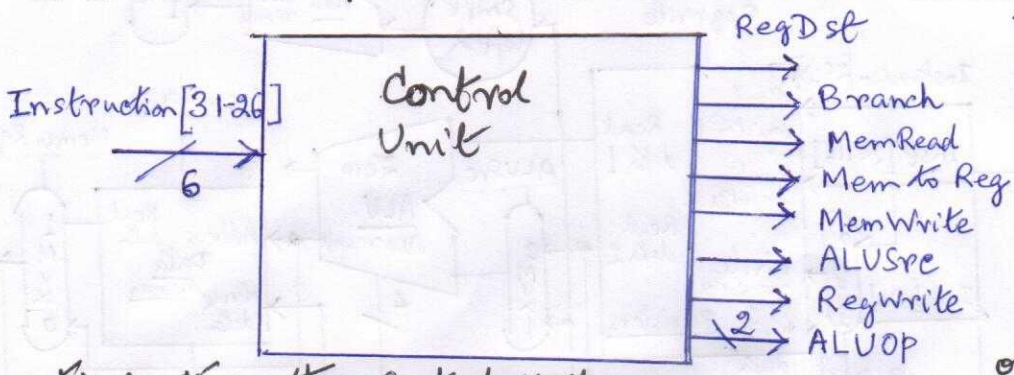
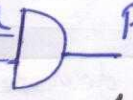


Figure 15: the control unit.

• The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control the multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU,



PCSrc (Mux select of Fig. 13)

The AND gate output controls the selection of the next PC.

< Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. >

• Before one tries to write a set of equations or a truth table for the control unit, one may define the control function informally. As the setting of the control lines depends only on the opcode, one defines whether each control signal should be 0, 1, or don't care (X) for each of the opcode values. Figure 16 (on page 3/11) defines how the control signals should be set for each opcode.

Instruction	RegDst	ALUSrc	MemtoReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

Figure 16: the setting of the control lines is completely determined by the opcode fields of the instruction. For the R-format instructions (add, sub, AND, OR, and slt) corresponding to the first row of the above table, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Moreover, an R-type instruction writes a register (RegWrite=1), but neither reads or writes data memory. When the Branch signal is 0, the PC is unconditionally replaced with PC+4; else, the PC is replaced by the branch target if the Zero o/p of ALU is also high.

The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of the table give the control signal settings for lw and sw. The ALUSrc and ALUOp fields are set to perform address calculation. The MemRead and MemWrite are set to perform memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch inst^n is similar to an R-format operation, as it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Note that MemtoReg field is irrelevant when the RegWrite signal is 0; since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X (for don't care). Don't cares can also be added to RegDst when RegWrite is 0. This type of don't cares must be added by the designer, since it depends on knowledge of how the datapath works.

P.T.O.

Operation of the Datapath

Before designing the control unit logic, it will be interesting to know how each instruction uses the datapath. One can show the flow of different instruction classes through the datapath. For this, the asserted control signals and active datapath elements should be highlighted. Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control lines should be highlighted if any constituent signal is asserted.

- Consider the operation of the datapath for an R-type instruction, such as `add $t1, $t2, $t3`.

Although everything occurs in one clock cycle, one can think of the following four steps to execute the instruction; these steps are ordered by the flow of information:

1) the instruction is fetched, and the PC is incremented
2) two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3) the ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.
4) the result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

- Similarly, one can consider the execution of a 'load word' instruction such as `lw $t1, offset($t2)`.

The load instruction may be thought as operating in five steps as:

1) an instruction is fetched from the instruction memory, and the PC is incremented.
2) a register ($t2) value is read from the register file
3) the ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset)
4) the sum from the ALU is used as the address for the data memory.
5) the data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1)

operation of the Datapath (cont.)

Finally, consider the branch-on-equal instruction, such as

beq $t1, $t2, offset

in similar manner. Though it operates much like an R-format inst^{n}, the ALU output is used to determine whether the PC is written with PC+4 or the branch target address. The 'branch on equal' inst^{n} is executed in the following four steps:

1) an instruction is fetched from the instruction memory, and the PC is incremented.

2) two registers, $t1 and $t2, are read from the register file.

3) the ALU performs a subtraction operation on the data values read from the register file. The value of PC+4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.

4) the Zero result from the ALU is used to decide which adder result to store into the PC.

Finalizing Control

Having seen how the instructions operate in steps, one can proceed with the control implementation. The control function can be precisely defined using the contents of Figure 16. One can create a truth table for each of the nine outputs (control lines), based on the binary encoding of the 6-bit opcodes (as inputs).

Figure 17 shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and one can implement it directly in gates.

Input or output	Signal Name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figure 17: the control function for the simple single-cycle implementation is completely specified by this truth table. The top six rows of the table gives combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings. Remember that Op[5:0] correspond to bits 31:26 of the instruction, which is the op field. The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs.

As one has a single-cycle implementation of most of the MIPS core instruction set, let us add the jump instruction to show how the basic datapath and control can be extended to handle other instructions.

Implementing Jumps

Field	000010	address
Bit positions	31:26	25:0

Figure 18: Instruction format for the jump instruction (opcode = 2). The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC+4 to the 26-bit address field in the jump instruction and adding 00 as the two low-order bits.

Though the jump inst^n looks similar to a branch inst^n, it computes the target PC differently, and is not conditional. Like a branch, the low-order 2 bits of a jump address are always 00_2. The next upper 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction. The upper 4 bits of the address, that should replace the PC, come from the PC of the jump instruction plus 4. Thus we can implement a jump by storing into the PC the concatenation of

i) upper 4 bits of the current PC+4 (these are bits 31:28 of the sequentially following instruction address)
ii) 26-bit immediate field of the jump instruction
iii) bits 00_2.

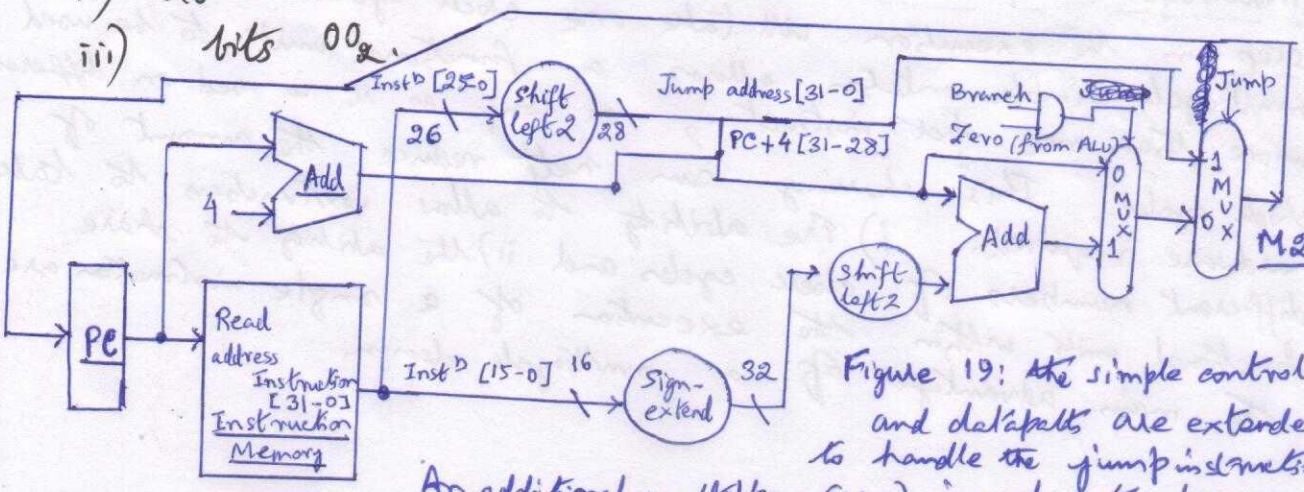


Figure 19: The simple control and datapath are extended to handle the jump instruction

An additional multiplexor (M2) is used to choose between the jump target (address) and either the branch target (address) or the sequential instruction (address) following the current one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left by 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC+4 as the high-order bits, thus producing a 32-bit address

P.T.O.

Why is a Single-Cycle Implementation not used in modern designs?

Although the single-cycle design works correctly, it is inefficient. Note that the clock cycle must have the same length for every instruction in this single-cycle design. The length of the clock period is determined by the longest possible path in the processor. This path is certainly a load (or store) word instruction, as it uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file.

Although the CPI (cycles per instruction) is 1, the overall performance of a single-cycle implementation will be quite poor, as the clock cycle is too long.

- For this small instruction set, the penalty for using the single-cycle design with a fixed clock cycle may be acceptable. However, if one tries to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design will not work well at all.

A Multicycle Implementation

Earlier, we divided execution of each instruction into a series of steps related to the functional unit operations that were needed. Using these steps, one can create a multicycle implementation. In a multicycle implementation, each step in the execution will take one clock cycle. The multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help reduce the amount of hardware required. i) The ability to allow instructions to take different numbers of clock cycles and ii) the ability to share functional units within the execution of a single instruction are the main advantages of a multicycle design.