The Processor

- One may examine an implementation of a processor that can execute the following instructions:
  1) memory-reference instructions load word (lw) and store word (sw)
  2) arithmetic-logical instructions add, sub, AND, OR and slt (set less than e.g. slt rd, rs, rt (set register rd to 1 if rs is less than rt, and 0 otherwise))
  3) control flow instructions branch equal (beq) and jump (j)
- The above subset (of instructions) does not include all the integer instructions (e.g. shift, multiply and divide are missing), nor does it include any floating-point instructions. However, one can create a datapath and design the control. <Implementation of the remaining instructions is similar>
- While studying the implementation, one can see how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI (cycles per instruction) for the computer.
- To implement every instruction (independent of the above three classes of instruction), the first two steps are identical:
  1) send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
  2) read one or two registers, using fields of the instruction to select the registers to read. For the 'load word' instruction, one needs to read only one register, but most other instructions require reading two registers
- After the above two steps, actions required to complete the instruction depend on the instruction class. For each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are mostly same, regardless of the exact instruction. Simplicity and regularity of the instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

P.T.O.

• For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison. After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction needs to access the memory in order to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from ALU or memory back into a register. Finally, for a branch instruction, one may need to change the next instruction address based on the comparison; else, the PC should be incremented by 4 to get the address of the next instruction.

• Figure 1 (or page 3/a) shows the high-level view of a MIPS implementation, focusing on various functional units and their interconnection. In spite of the figure showing most of the flow of data through the processor, it omits two important aspects of instruction execution. First, at several places, data going to a particular unit is shown as coming from two different sources. For example, value written into PC can from one of two adders; also, data written into the register file can from either ALU or data memory; similarly, the second input to the ALU can come from a register or the immediate field of the instruction. In practice, one cannot wire together these data lines. One requires a multiplexor (or data selector). Second (omission) in Fig. 1 is that several of the units must be controlled depending on the type of instruction. For example, the data memory must read on a load and write on a store. The register file must be written only on a load or on an arithmetic-logical instruction. Also, the ALU must perform one of several operations. Like the multiplexors, control lines are set on the basis of various fields in the instruction to direct these operations.

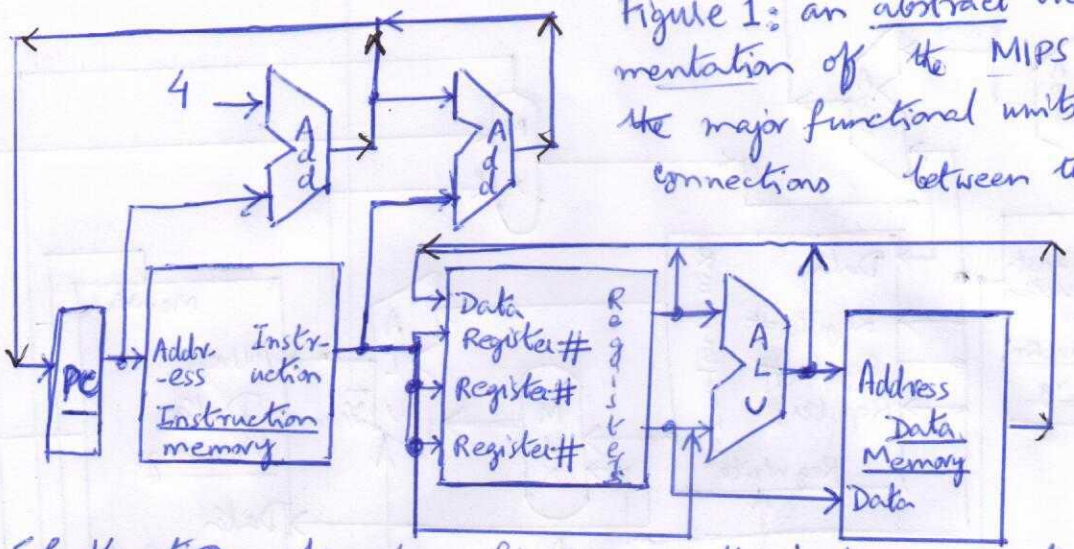


Figure 1: an abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

[Explanation of above figure: all instructions start by using the program counter (PC) to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. On fetching the register operands, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result of the ALU must be written to a register. If the instruction is a load or a store, the ALU result is used as an address to either load a value from memory into the registers, or store a value from the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed), or from an adder that increments the current PC by 4.

- Figure 2 (drawn on the back side) shows the datapath (of Fig. 1) with three required multiplexors added, as well as control lines for the major functional units. A control unit, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors. The third MUX, which determines whether PC+4 or the branch destination address is written into the PC, is set based on Zero output of the ALU (which is used to perform comparison of a beq instruction.

P.T.O.

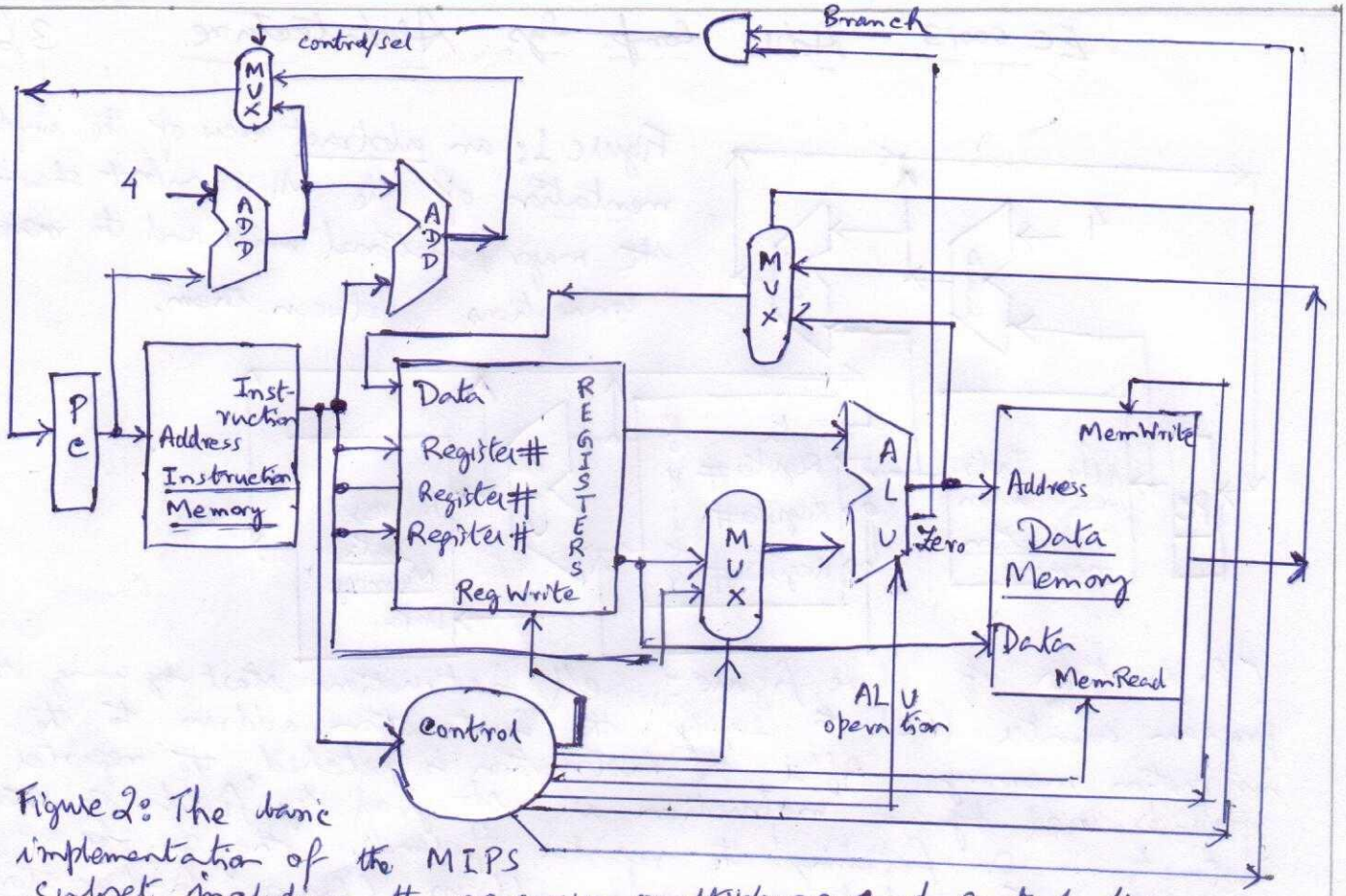


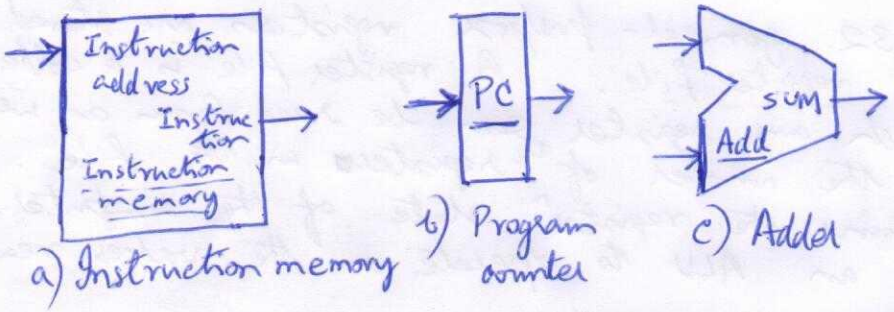
Figure 2: The basic implementation of the MIPS subset, including the necessary multiplexors and control lines.

[Explanation of above figure: the top multiplexor (MUX) controls what value replaces the PC (PC+4 or the branch destination address); this MUX is controlled by the gate that 'ANDs' together the Zero output of the ALU and a control signal which indicates that the instruction is a branch. The middle MUX, whose output returns to the register file, is used to steer the o/p of the ALU (in case of an arithmetic-logical inst^n) or the o/p of the data memory (in case of a load) for writing into the register file. Finally, the bottommost MUX is used to determine whether the second ALU i/p is from registers (for an arithmetic-logical inst^n or a branch) or from the offset field of the instruction (for a load or store).

The added control lines determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation.

Building a Datapath

- To start a datapath design, one needs to examine the major components required to execute each class of MIPS instructions. One must know the datapath elements needed by each instruction
- datapath element : a unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.
- program counter (PC) : the register containing the address of the instruction in the program being executed.



a) Instruction memory b) Program counter c) Adder

Figure 3: two state (sequential) elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

As seen above, one requires i) Instruction memory : a memory unit to store the instructions of a program and supply instructions, given an address, ii) Program counter (PC) : a register that holds the address of the current instruction, and iii) an Adder to increment the PC to the address of the next instruction.

- To execute any instruction, one must start by fetching the instruction from memory. To prepare for executing the next instruction, one must also increment the PC so that it points at the next instruction, 4 bytes later.

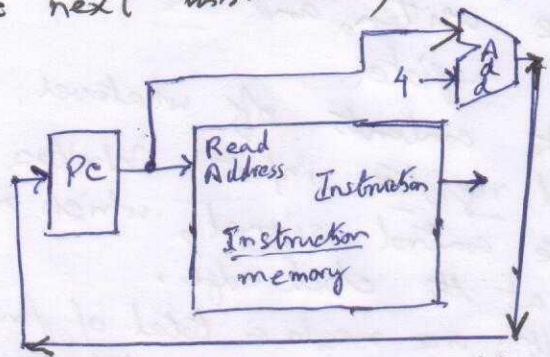


Figure 4: a portion of the datapath used for fetching instructions and incrementing the program counter, to obtain the address of the next sequential instruction

- Let us now consider the R-format instructions, all of which read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. One calls these

instructions either R-type instructions or arithmetic-logical instructions (since they perform arithmetic or logical operations). This instruction class includes add, sub., AND, OR and slt. A typical instance of such an instruction is

add $t1, $t2, $t3, which reads $t2 and $t3, computes sum of their contents, and writes the sum to $t1

Name	Fields		'Shamt': shift amount for shift instructions				Comments
Field size	6-bits	5 bits	5 bits	5 bits	5 bits	6 bits	all MIPS instructions are 32-bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format

Fig: MIPS Instruction format

- The processor's 32 general-purpose registers are stored in a structure called a register file. A register file is a collection of registers in which any register can be read from or written to by specifying the number of the register in the file. The register file contains the register state of the computer. In addition, one needs an ALU to operate on the values read from the registers.
- R-format instructions have three register operands; so, one needs to read two data words from the register file, and write one data word into the register file for each instruction.
  1) For each data word to be read from registers, one needs an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers.
  2) To write a data word, one needs two inputs: one to specify the register number to be written, and one to supply the data to be written into the register.
- The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur ~~on~~ at the clock edge.
- Figure 5 (on page 3/4) shows the result; one needs a total of four inputs (three for register numbers and one for data) and two outputs (both for data). Register number inputs are 5 bits wide to specify one of 32 registers ($2^5 = 32$), while the data i/p & o/p busses are 32 bits wide each.

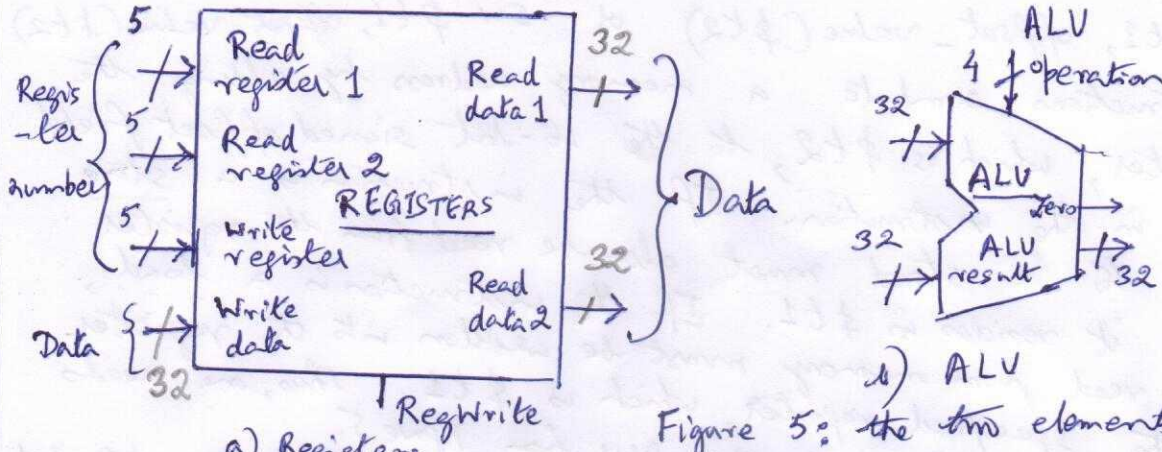


Figure 5: the two elements needed to implement R-format ALU operations

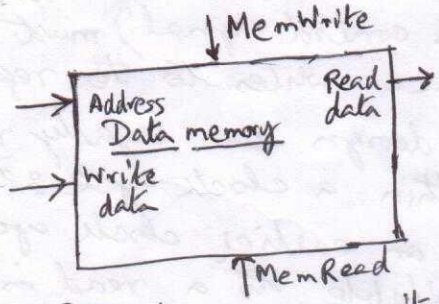
are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. The writes are edge-triggered, so that all the write inputs (i.e value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, the present design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The operation to be performed by the ALU is controlled with the "ALU operation" signal, which will be 4 bits wide; the ALU has the Zero detection output, which may be used to implement Branches. This ALU has also an overflow output.

Next, let us consider the MIPS load word and store word instructions, which have the general form

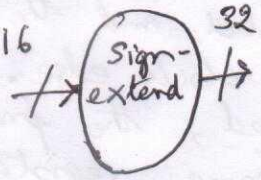
lw $t1, offset_value ($t2) or sw $t1, offset_value ($t2).

These instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in $t1. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is $t1. Thus, one needs both the register file and the ALU from Figure 5.

Moreover, one will need a unit to sign-extend the 16-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory.



a) Data memory unit



b) Sign extension unit

Figure 6: the two units needed to implement loads and stores, in addition to the register file and ALU (of Fig 5) are the data memory unit and the sign extension unit.

The memory unit is a sequential element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, though only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems.

The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output. One assumes that the data memory is edge-triggered for writes.

branch on equal (beq) (conditional branch instruction)

e.g. beq $s1, $s2, 25 meaning: if ($s1 == $s2) then go to PC+4+100 equal test ; PC-relative branch

The "beq" instruction has three operands, two registers that are compared for equality, and a 16-bit offset (signed) offset field to compute the branch target address relative to the branch instruction address. To implement the instruction beq $t1, $t2, offset one must compute the branch target address by adding the sign-extended offset field of the instruction to the PC. One should consider the two details in the definition of branch instructions.

i) the instruction set architecture (ISA) specifies that the base for the branch address calculation is the address of the instruction following the branch. As one computes PC+4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.

ii) the ISA also states that the offset field is shifted left by 2 bits so that it is a word offset; (as one word ≡ 4 bytes) this shift increases the effective range of the offset field by a factor of 4.

< to deal with the latter complication, one needs to shift the offset field by 2>

• One must determine whether the next instruction is the instruction that follows sequentially, or the instruction at the branch target address. When condition is true (i.e. operands are equal), the branch target address becomes the new PC, and one says that the branch is taken. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case one says that the branch is not taken.

• So, the branch datapath must do two operations: compute the branch target address and compare register contents. Figure 7 (drawn over-leaf) shows the structure of the datapath segment that handles branches.

P.T.O.

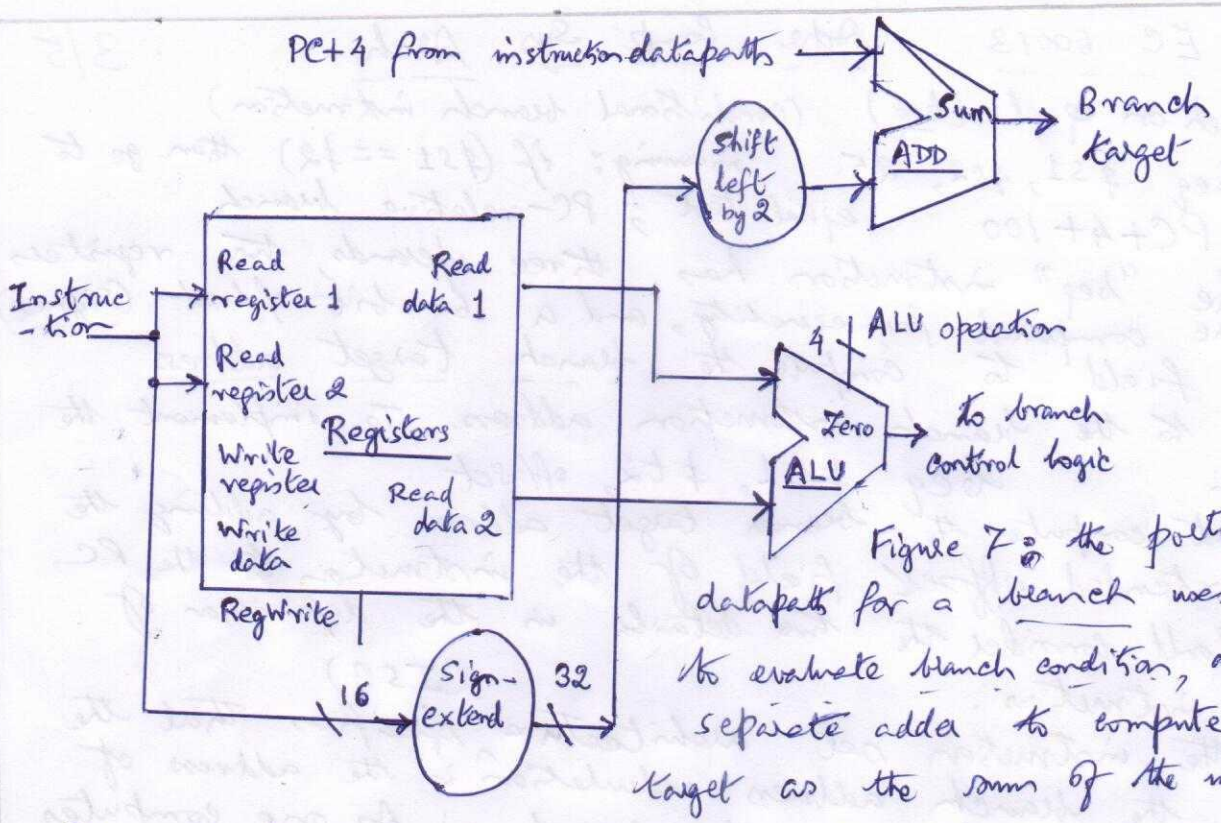
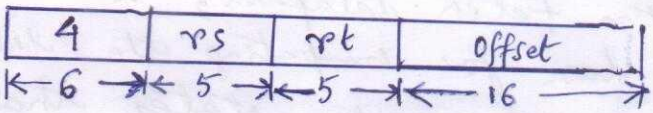


Figure 7: the portion of a datapath for a branch uses the ALU to evaluate branch condition, and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left by 2 bits.

Branch on equal
beq rs, rt, label



N.B. In the above figure, the unit labeled "Shift left 2" is just a routing of the signals bet^n i/p and o/p that adds 00_2 to the low-order end of the sign-extended offset field (no actual shift hardware is needed). Control logic decides whether the incremented PC or branch target should replace the PC, based on the Zero o/p of the ALU

Delayed branch: In the MIPS instruction set, branches are delayed; it means that the instruction immediately following the branch (instruction) is always executed, regardless of whether the branch condition is true or false. When the condition is true, a delayed branch first executes the inst^n immediately following the branch in sequential instruction order before jumping to the specified branch target address.

< Motivation for delayed branches: arises from how pipelining affects branches; for simplicity, one generally ignores delayed branches now and implements a nondelayed beq instruction >.

Creating a Single Datapath

- Having examined datapath components needed for individual instruction classes (viz. memory reference, arithmetic-logic, control flow), one can combine them into a single datapath and add the control to complete the implementation.
- the simplest datapaths attempts to execute all instructions in one clock cycle. < Thus, no datapath resource can be used more than once per instruction; so any element needed more than once must be duplicated. One therefore needs a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows. >
- To share a datapath element between two different instruction classes, one may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

Building a Datapath

- The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The main differences are as follows :-

1) the arithmetic-logical instructions use the ALU, and the inputs coming from the two registers. The memory instructions can also use the ALU to do address calculation, although the second input is the sign-extended 16-bit offset field from the instruction.

2) the value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load instruction).

Problem: to show how to build a datapath for the operational portion of the memory-reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

Solution: to create a datapath with only a single register file and a single ALU, one must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. Figure 8 shows the operational portion of the combined datapath.

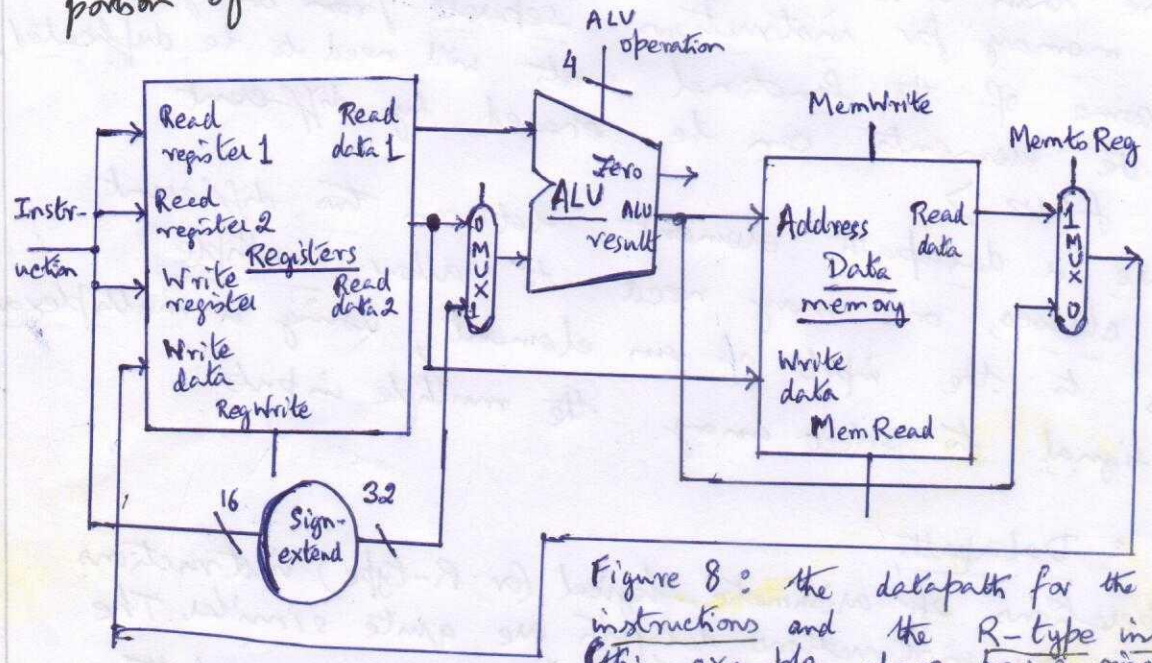


Figure 8: the datapath for the memory instructions and the R-type instructions (this example shows how a single datapath can be assembled from the pieces in Figures 5 and 6 by adding multiplexors)

- Now one can combine all the pieces to make a simple datapath for the core MIPS architecture by adding the datapaths for instruction fetch (Figure 4), the datapaths from R-type and memory instructions (Fig. 8), and datapaths for branches (Fig. 6). Fig. 9 shows the datapaths one obtains by composing the separate pieces. The branch instruction uses the main ALU for comparing the register operands, so one must keep the adder from Fig. 6 for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.