SINGLE CYCLE 16-BIT MIPS PROCESSER

Special assignment

*Submitted in Partial Fulfillment of the Requirements for the Degree*

Of

**Bachelor of Technology**

In

**Electronics & Communication Engineering**

**Submitted By:**

KASHYAP ADODARIYA (17BEC004)

****

**Department of Electronics & Communication Engineering**

Institute of Technology, Nirma University,

Ahmedabad – 382481

April – 2019

* Instruction Execution
* PC → instruction memory, fetch instruction
* Register numbers → register file, read registers
* Use ALU to calculate

1. Arithmetic result
2. Memory address for load/store
3. Branch target address

* Access data memory for load/store
* PC ← target address or PC + 2

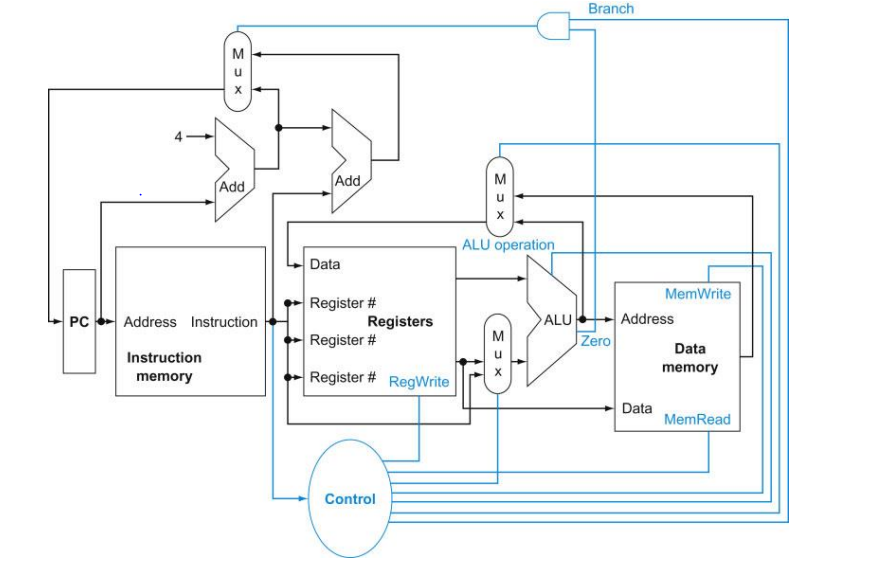


Figure 1: single cycle MIPS datapath

* All instruction classes, except jump, use the arithmetic-logic unit (ALU) after reading the registers.
* The data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a register or the immediate file of the instruction.
* The ALU must perform one of several operations. Like the multiplexors, control lines that are set on the basis of various filed in the instruction direct these operations.
* three required multiplexors added, as well as control lines for the major function 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 multiplexors which determined whether PC + 4 or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a beq instruction

R-type

* R-format instructions all read two registers, perform an ALU operation on the contents of the register, and write the result to a register.
* We call these instructions either R-type instructions or arithmetic-logical instructions. This instruction class includes add, sub, AND, OR, and slt.
* Recall that a typical instance of such an instruction is: add $t1, $t2, $t3
* R-format Instructions:
* Read two register operands
* Perform arithmetic/logical operation
* Write register result

Load/Store Instructions

* lw $t1, offset ($t2) # $t1 = Memory [$t2 + offset]
* sw $t1, offset ($t2) # Memory [$t2 + offset] = $t1
* Load/Store Instructions:
* Read register operands
* Calculate address using 8-bit offset
* Load: Read memory and update register
* Store: Write register value to memory

Branch-on-Equal Instruction

* The beq branch instruction has three operands, two register that are compared for equality, and a 16-bit offset used to compute the branch target address relative to the branch instruction address.

beq $t1, $t2, offset # if ($t1 == $t2) go to (PC + 2) + (offset \* 2)

* Branch Instruction
* Read register operands
* Compare operands
* Use ALU, subtract and check Zero output
* Calculate target address
* Sign-extend displacement
* Shift left 2 places (word displacement)
* Add to PC + 2: Already calculated by instruction fetch

ALU Control

control signals and generates a write signal for each state element

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ALU control | | | | |
| Instruction | ALU op | Function | ALU cnt | ALU Operation |
| Addi, lw, sw | 11 | xxxx | 000 | ADD |
| BEQ | 01 | xxxx | 001 | SUB |
| R-type : Add | 00 | 00 | 000 | ADD |
| R-type : Sub | 00 | 01 | 001 | SUB |
| R-type : AND | 00 | 02 | 010 | AND |
| R-type : OR | 00 | 03 | 011 | OR |
| R-type : Slt | 00 | 04 | 100 | Slt |
| I-type : Slti | 10 | xxxx | 100 | Slt |

Table 1: ALU Control

Designing the Main Control Unit

The first step in designing the main control unit is to identify the fields of each instruction and the required control lines to implement the datapath shown in Figure.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | Fields | | | | |
| Field Size | 3-bit | 3-bit | 3-bit | 3-bit | 4-bit |
| R-format | Op | Rs | Rt | Rd | Function |
| I-format | Op | Rs | Rt | Address/immediate | |
| J-format | Op | Target address | | | |

Table 2: Instruction set architecture

Op: basic operation of the instruction

Rs: first operant register

Rt: second operant register

Rd: destination operand register

Function: select the special variant of opcode

Address/immediate: offset for load/store instruction(2^8bit)

Target address: offset for J-type instruction

* RegDst
  + Deasserted: Register destination number for the Write register is taken from bits 20-16 (rt field) of the instruction
  + Asserted: Register destination number for the Write register is taken from bits 15-11 (rd field) of the instruction
* RegWrite
  + Deasserted: No action
  + Asserted: Register on the WriteRegister input is written with the value on the WriteData input
* ALUSrc
  + Deasserted: The second ALU operand is taken from the second register file output (ReadData 2)
  + Asserted: the second ALU operand is the sign-extended, lower 8 bits of the instruction
* PCSrc
  + Deasserted: PC is overwritten by the output of the adder (PC + 2)
  + Asserted: PC overwritten by the branch target address
* MemRead
  + Deasserted: No action
  + Asserted: Data memory contents designated by address input are present at the ReadData output
* MemWrite
  + Deasserted: No action
  + Asserted: Data memory contents designated by address input are present at the WriteData input
* RegWrite
  + Deasserted: The value present at the WriteData input is output from the ALU
  + Asserted: The value present at the register WriteData input is taken from data memory

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Control signal | | | | | | | | | |
| Instruction | Reg. DSt. | ALU Src | Memto Reg | Reg Write | MemRead | Mem Write | Branch | ALUop | Jump |
| R-type | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 00 | 0 |
| LW | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 11 | 0 |
| SW | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 11 | 0 |
| Addi | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 11 | 0 |
| Beq | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 01 | 0 |
| J | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 1 |
| Jal | 2 | 0 | 2 | 1 | 0 | 0 | 0 | 00 | 1 |
| Slti | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 10 | 0 |

Table 3:Control signal

**List of instruction:**

1. Add: R[rd] = R[rs] + R[rt]
2. Subtract: R[rd] = R[rs] - R[rt]
3. And: R[rd] = R[rs] & R[rt]
4. Or: R[rd] = R[rs] | R[rt]
5. SLT: R[rd] = 1 if R[rs] < R[rt] else 0
6. Jr: PC=R[rs]
7. Lw: R[rt] = M[R[rs]+SignExtImm]
8. Sw: M[R[rs]+SignExtImm] = R[rt]
9. Beq: if(R[rs]==R[rt]) PC=PC+1+BranchAddr
10. Addi: R[rt] = R[rs] + SignExtImm
11. J: PC=JumpAddr
12. Jal: R [7] =PC+2; PC=JumpAddr
13. SLTI: R[rt] = 1 if R[rs] < imm else 0

**Datapath Operation**

Recall that there are three MIPS instruction formats -- R, I, and J. Each instruction causes slightly different functionality to occur along the datapath, as follows.

**R-format Instruction**.

Execution of an R-format instruction (e.g., add $t1, $t0, $t1) using the datapath developed the following steps:

1. Fetch instruction from instruction memory and increment PC
2. Input registers (e.g., $t0 and $t1) are read from the register file
3. ALU operates on data from register file using the funct field of the MIPS instruction (Bits 5-0) to help select the ALU operation
4. Result from ALU written into register file using bits 15-11 of instruction to select the destination register (e.g., $t1).

Note that this implementation sequence is actually combinational, because of the single-cycle assumption. Since the datapath operates within one clock cycle

**Load/Store Instruction.**

 Execution of a load/store instruction using the datapath developed the following steps:

1. Fetch instruction from instruction memory and increment PC
2. Read register value (e.g., base address in $t2) from the register file
3. ALU adds the base address from register $t2 to the sign-extended lower 8 bits of the instruction (i.e., offset)
4. Result from ALU is applied as an address to the data memory
5. Data retrieved from the memory unit is written into the register file, where the register index is given by $t1.

**Branch Instruction**

Execution of a branch instruction (e.g., beq $t1, $t2, offset) using the datapath developed in the following steps:

1. Fetch instruction from instruction memory and increment PC
2. Read registers (e.g., $t1 and $t2) from the register file. The adder sums PC + 4 plus sign-extended lower 16 bits of offset shifted left by two bits, thereby producing the branch target address (BTA).
3. ALU subtracts contents of $t1 minus contents of $t2. The Zero output of the ALU directs which result (PC+4 or BTA) to write as the new PC.

#### Limitations of the Single-Cycle Datapath

The single-cycle datapath is not used in modern processors, because it is inefficient. The critical path (longest propagation sequence through the datapath) is five components for the load instruction. The cycle time tc is limited by the settling time ts of these components. For a circuit with no feedback loops, tc > 5ts. In practice, tc = 5kts, with large proportionality constant k, due to feedback loops, delayed settling due to circuit noise, etc. it is possible to compute the required execution time for each instruction class from the critical path information. The result is that the Load instruction takes 5 units of time, while the Store and R-format instructions take 4 units of time. All the other types of instructions that the datapath is designed to execute run faster, requiring three units of time.

The problem of penalizing addition, subtraction, and comparison operations to accomodate loads and stores leads one to ask if multiple cycles of a much faster clock could be used for each part of the fetch-decode-execute cycle. In practice, this technique is employed in CPU design and implementation, as discussed in the following sections on multicycle datapath design. we will show that datapath actions can be interleaved in time to yield a potentially fast implementation of the fetch-decode-execute cycle that is formalized in a technique called pipelining.