Processor: Datapath and Control

Introduction

- Clock cycle time and number of CPI are determined by processor implementation
- Datapath and control
- Effect of different implementation choices on clock rate and CPI
- Implementation overview
  - Two identical first step for *every* instruction
    1. Send PC to memory and fetch the instruction from that location
    2. Read one or two registers, selected by fields of instruction opcode (one register for `lw`, two for most of the other instructions)
  - Perform the actions to accomplish the instruction
    * Actions are largely the same, independent of exact opcode
    * Holds for each instruction class: memory reference, arithmetic-logic, and branch
  - All instruction classes use ALU after reading registers
    * Memory reference instructions use ALU for address calculations
    * Arithmetic-logic instructions use ALU for operation execution
    * Branch instructions use ALU for comparison
  - Post-ALU execution of instructions
    * Memory-reference instruction needs to access memory for load or store
    * Arithmetic-logic instruction must write data to a register
    * Branch instruction needs to change the PC for next instruction address based on comparison
  - Figure 5.1
    * High level view of a MIPS implementation
- Logic convention and clocking
  - Designer may need to change the mapping between a logically true or false signal and the high or low voltage level
  - In some parts of a design, a signal that is logically asserted may be electrically low while in other parts, an electrically high signal is asserted

    asserted indicates a signal that is logically high

    assert indicates a signal that should be driven logically high
  - Functional unit in MIPS implementation has two different logic elements
    1. Combinational elements
       * Elements that operate on data values
       * Output depends only on the current inputs
       * No internal storage
       * Exemplified by circuitry to perform arithmetic operations
    2. State elements
       * Elements that contain state, or internal storage
       * Characterize the state of the machine
       * If machine loses power but these elements can be restored, we can restart the machine from the state at which the power was lost
       * Instructions and data memories, as well as registers

        * At least two inputs and one output
        * Inputs are data value to be written into the element and the clock to determine when the data value is written
        * Output is the value that was written during an earlier clock cycle
        * One of the logically simplest state elements is a D-type flip flop that has exactly two inputs and one output
        * Other state elements used in MIPS implementation are memories and registers
    - Logic components containing state are called *sequential* because their output depends on both internal state and inputs

- Clocking methodology
    - Defines when signals can be read or written
    - Concurrency issue; read at the same time as write
    - Edge-triggered clocking methodology
        * Any values are to be updated only on a clock edge
        * State elements update their stored value on the clock edge
        * Figure 5.2
        * Any collection of combinational logic must have its inputs coming from a set of state elements and its outputs written into a set of state elements
        * Inputs are values written in a previous clock cycle
        * Outputs are values to be used in a following clock cycle
        * Length of the clock cycle is the time taken by the system to go from one element to the other
        * If a state element is not updated on every edge, an explicit write control signal is required
            · State element is updated only when the write control signal is asserted and a clock edge occurs
    - Figure 5.3
        * Read, operate, and write can be achieved in the same clock cycle
        * No feedback within a single clock cycle
    - All state and logic elements have 32-bit wide inputs and outputs
        * Buses are signals wider than 1 bit
        * Several buses can be combined to make a wider bus

- MIPS subset implementation
    - Simple implementation using a single clock cycle for every instruction
    - Not practical
        * Does not allow different instruction classes to take different number of clock cycles that may be shorter

Building a datapath

- Major components required to execute each class of MIPS instruction
    - State element memory unit
        * Holds and supplies instructions given an address
    - State element PC
        * Holds the address of next instruction to be executed
    - Combinational logic adder
        * Makes PC point to the next instruction's address

    - Built from the ALU
  - Figure 5.4
- Instruction execution
  - Fetch the instruction from memory
  - Increment PC by 4
  - Figure 5.5
- *R*-format instructions
  - Includes arithmetic-logic instructions, such as `add`, `sub`, `slt`, `and`, and `or`
  - Read two registers, perform an operation, and write result to a third register
  - Registers are stored in a structure called *register file*
    - Register file has two read ports and one write port
    - For each data word to be read from [written to] register file, we need
      - Register number to be read from [written to]
      - Output from [Input to] the register file to carry the value being read [written]
    - Writes are asserted by a write control signal, for a write to occur at a clock edge
    - Figure 5.6
    - Four inputs – three to identify registers, one for data
      - Inputs to identify registers are 5-bit wide
      - Data input is 32-bits
    - Two 32-bit outputs for data
  - ALU is controlled by a 3-bit signal
    - 1-bit `bnegate` and 2-bit `operation`
    - Figure 4.19
    - Signals are given by

ALU control lines	Function
0 00	`and`
0 01	`or`
0 10	`add`
1 10	`sub`
1 11	`slt`

    - Two 32-bit inputs and 1 32-bit output
  - Datapath for *R*-type instructions
    - Figure 5.7
  - `lw` and `sw` instructions
    - General format: `lw $t1, offset ( $t2 )`
    - Compute the memory address by adding the base register (`$t2`) to the 16-bit signed offset field
    - For `sw`, value to be stored must be read from register file (`$t1`)
    - For `lw`, value read from memory must be stored to register file in `t1`
    - Need the register file and ALU
    - Also need a unit to sign extend 16-bit offset to 32-bit signed value, and a data memory unit (Figure 5.8)
    - Data memory
      - Must be written on `sw`
      - Must have both read and write control signal

            - Must have an address input
            - Must have a data input
        - Figure 5.9
    - `beq` instruction
        - `beq  $t1, $t2, offset`
        - Three operands: two registers to compare for equality and a 16-bit offset to compute branch address relative to current instruction address
        - Compute the branch address by adding sign-extended offset to PC
            - Base of branch address is the address of instruction following the branch
            - Offset field is shifted left by 2-bits to make it a word offset, increasing the effective range of offset by a factor of 4
            - This also requires the offset field to be shifted by 2 bits
        - Must also determine the next instruction address (very next or offset'ed) – contents of PC
        - Datapath has to compute the target address and compare two registers
        - Figure 5.10
        - Equality indicated by a zero asserted out of ALU, achieved by subtracting one register from another
        - Jump operates by replacing the lower 28 bits of PC with lower 26 bits of instruction shifted left by 2 bits

Simple implementation scheme

- Creating a single datapath
    - Executing all instructions in a single clock cycle
        - No datapath resource can be used more than once in an instruction
        - Any element needed more than once must be duplicated
        - Instruction memory must be separated from data (separate datapath resources)
    - Sharing datapath elements between different instruction classes
        - Need multiple connections to the input of an element, selected by a control signal
        - Commonly achieved by a multiplexor
- Composing datapaths
    - Key differences between *R*-type datapath and memory instruction datapath
        - Second input to ALU is either a register (*R*-type) or sign-extended lower half of instruction (memory)
        - Value stored in destination register comes from ALU (*R*-type) or memory (`lw`)
    - Problem: Combine the two datapaths using multiplexors, without duplicating the common functional units
    - Solution
        - Must support two separate sources for second ALU input
        - Support two different sources for data stored in register files
        - Figure 5.11
    - Adding instruction fetch portion to the datapath
        - Figure 5.12
        - Separate instruction and data memory because of single cycle execution
        - Requires an adder (to increment PC) and an ALU (to execute instruction in the same clock cycle)
    - Adding datapath for branches

  - Figure 5.13
  - Uses main ALU for comparing registers
  - Keep the adder to compute the branch target address
  - Additional multiplexor to select between target address and sequentially following address for PC

- Control unit
  - Must be able to take inputs
  - Must be able to generate
    - A write signal for each state element
    - Selector control for each multiplexor
    - ALU control

- ALU control
  - Three control inputs (bits) – one for `bnegate` and two for operation
  - Only five of possible eight combinations are used

ALU control input	Function
000	`and`
001	`or`
010	`add`
110	`sub`
111	`slt`

  - Depending on instruction class, ALU has to perform one of these five functions
  - For `lw` and `sw`, ALU computes memory address by addition
  - For *R*-type instructions, ALU does one of the five functions based on the value of 6-bit `funct` field in low-order bits of opcode
  - For branch, ALU performs a subtraction
  - Control unit for the 3-bit ALU control input
    - Input from `funct` field of opcode and a 2-bit control field called ALUOp
    - ALUOp indicates operation

Bits	Operation
00	Add for load and store
01	Subtract for `beq`
10	Determined by operation encoded in `funct` field

    - Output of ALU control is a 3-bit signal (one of five combinations) to control the ALU
  - Setting ALU control inputs based on 2-bit ALUOp control and 6-bit `funct` field

Instruction type	ALUOp	Instruction operation	`funct` field	Desired ALU action	ALU Control input
Memory	00	`lw`	xxxxxx	Add	010
Memory	00	`sw`	xxxxxx	Add	010
Branch	01	`beq`	xxxxxx	Subtract	110
R-type	10	`add`	100000	Add	010
R-type	10	`sub`	100010	Subtract	110
R-type	10	`and`	100100	AND	000
R-type	10	`or`	100101	OR	001
R-type	10	`slt`	101010	set on less than	111

    Notice the relationship between ALUOp and instruction type
  - Hierarchy of multiple decoding levels

  - Main control unit generates ALUOp bits
  - ALU control unit uses ALUOp bits to generate actual signals
  - Multiple levels reduce size of control unit
  - Potential increase in control unit speed
- Mapping from ALUOp bits and `funct` to 3-bit ALU operation
  - `funct` is used only when ALUOp is 10
  - Create a truth table of the small number of possibilities
  - Only show those inputs for which the output is defined

ALUOp		`funct`						Operation
ALUOp1	ALUOp2	F_5	F_4	F_3	F_2	F_1	F_0	
0	0	x	x	x	x	x	x	010
x	1	x	x	x	x	x	x	110
1	x	x	x	0	0	0	0	010
1	x	x	x	0	0	1	0	110
1	x	x	x	0	1	0	0	000
1	x	x	x	0	1	0	1	001
1	x	x	x	1	0	1	0	111

- The truth table can be optimized (as above) and turned into hardware circuitry using standard algorithms

- Designing the main control unit
  - Fields of instructions and control lines needed for datapath
  - Instruction formats
    - *R*-type instruction

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

    - Memory instruction

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

    - Branch instructions

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

  - Observations
    - Opcode is always in bits 31-26 (high-order 6 bits); refer to it as op[5-0]
    - Two registers to be read are always specified by `rs` and `rt` in bits 25-21 and 20-16 (for all instructions except load)
    - Base register for load and store is always `rs`, bits 25-21
    - 16-bit offset for branch, load, and store is always in low-order 16 bits (15-0)
    - Destination register is in one of two places:
      - For load, it is in bits 20-16 (`rt`)
      - For *R*-type instructions, it is in bits 15-11 (`rd`)
      - No destination for other instructions (store and branch)
      - Add a multiplexor to select the field of instruction for register number
  - Add extra multiplexor and instruction labels to simple datapath
  - Figure 5.17
    - All multiplexors have two inputs, they require one control line
  - Effect of seven control signals

    RegDst Register destination number for write register

Deasserted: `rt` field, bits 20-16
Asserted: `rd` field, bits 15-11

RegWrite Register on the write register input
Deasserted: None
Asserted: Write value on write data input

ALUSrc Second ALU operand
Deasserted: Second register file output (read data 2)
Asserted: Sign extended

PCSrc Program counter value
Deasserted: PC + 4
Asserted: Branch target

MemRead Data memory contents
Deasserted: None
Asserted: Put the contents specified by address input on read data output

MemWrite Data memory contents
Deasserted: None
Asserted: Write into memory on specified address the data on write data input

MemtoReg Data into register write input
Deasserted: From ALU
Asserted: From data memory

- Setting the control signals
  - All signals except PCSrc can be set based on opcode field
    - PCSrc control line is set if instruction is branch on equal (determined by control unit) and ALU output is zero
    - AND the signal from control unit with zero signal from ALU
  - A total of nine control signals
    - Seven above and two ALUOp
    - Set on the basis of 6-bit opcode field input to control unit
    - Figure 5.19 – Datapath with control unit and control signals
  - Setting of control lines
    - Figure 5.20
    - Completely determined by opcode field

- Operation of datapath
  - Flow of three different instruction classes through datapath, with asserted control signals and active datapath elements highlighted
    1. *R*-type instruction (`add $t1, $t2, $t3`), executed in four steps
       (a) Fetch instruction from memory and increment PC (Figure 5.21)
       (b) Read two registers (`$t2` and `$t3`) from register file (Figure 5.22)
       (c) Operate on data read from register file, using `funct` bits (5–0) to generate ALU function (Figure 5.23)
       (d) Write the result from ALU to register file (bits 15–11) (Figure 5.24)
       - Implementation is combinational (not a series of four steps)
       - Datapath operates in a single clock cycle
       - Signals within datapath can vary unpredictably during clock cycle
    2. Memory instruction (`lw $t1, offset ( $t2 )`) – five steps
       (a) Fetch instruction from memory and increment PC

   (b) Read register value (`$t2`) from register file
   (c) Add the value read and sign-extended offset from lower 16-bits of instruction
   (d) Use the sum (from ALU computed above) as address in data memory
   (e) Write data from memory unit to register file; register destination in bits 20–16 (`$t1`)
   * Figure 5.25
3. Branch-on-equal instruction (`beq $t1, $t2, offset`)
   (a) Fetch instruction from memory and increment PC
   (b) Read registers `$t1` and `$t2` from register file
   (c) Use ALU to perform a subtract on data values read from registers; add `offset` (sig-extended and shifted lower 16-bits) to current value of PC (already incremented) to get branch target address
   (d) Use `zero` result from ALU to decide the address to be stored in PC
   * Figure 5.26

- Finalizing control
  - Defined using Figure 5.20
  - Output: Control lines
  - Input: 6-bit opcode (bits 5–0)
  - Encoding for input

Name	Opcode in decimal	Opcode in binary					
		op5	op4	op3	op2	op1	op0
R-format	0_{10}	0	0	0	0	0	0
`lw`	35_{10}	1	0	0	0	1	1
`sw`	43_{10}	1	0	1	0	1	1
`beq`	4_{10}	0	0	0	1	0	0

  - Control function truth table is given by

	Signal	*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
	`ALUOp2`	0	0	0	1

    * Observe the outputs being asserted (`RegWrite` for two different combinations of inputs)
    * *R*-format instruction can be identified by $\overline{op5} \cdot \overline{op2}$, but a more complete truth table will require more inputs to be identified

- Example: Implementing jumps
  - Jump instruction (opcode 2) format is given by

Field	2	address
Bit positions	31-26	25-0

  - `j` computes target PC differently and is not conditional
  - The low-order two bits are always 00
  - Next 26-bits come from the immediate field shown above
  - The upper 4 bits come from the current value of PC (incremented to point to next instruction)
  - Figure 5.29
  - Additional control signal for jump asserted only when opcode is 2

- Inefficiency of single-cycle implementation
  - Clock cycle must have the same length for every instruction in single cycle design
  - Clock cycle is determined by the longest possible path in the machine
  - For us, the longest path is the load instruction
    * Uses five functional units one after the other: instruction memory, register file, ALU, data memory, and register file
  - Several instructions could fit in a shorter clock cycle

- Example: Performance of single-cycle machine
  - Assume the operation time for major functional units as

Memory unit	2ns
ALU and adders	2ns
Register file (read or write)	1ns

  - Assume no delay in multiplexors, control unit, PC access, sign extension, and other wires
  - Compare the performance of following implementations
    1. Every instruction in one clock cycle of fixed length
    2. Every instruction in one clock cycle of variable length clock, with clock being only as long as needed
  - Assume a mix of instructions as 24% loads, 12% stores, 44% ALU, 18% branches, and 2% jumps
  - CPU execution time is computed as

$$\text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

  - For us, the CPI is 1, and hence

$$\text{CPU execution time} = \text{Instruction count} \times \text{Clock cycle time}$$

  - Critical (longest) path for different instruction classes

Instruction	Functional units used				
ALU type	Fetch	Reg. access	ALU	Reg. access	
Load word	Fetch	Reg. access	ALU	Mem. access	Reg. access
Store word	Fetch	Reg. access	ALU	Mem. access	
Branch	Fetch	Reg. access	ALU		
Jump	Fetch				

  - Compute the required length of each instruction class

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
ALU type	2	1	2	0	1	6ns
Load word	2	1	2	2	1	8ns
Store word	2	1	2	2		7ns
Branch	2	1	2			5ns
Jump	2					2ns

- Clock cycle is the longest path, or 8ns
- Variable clock time will be between 2 and 8ns. Average will be given by

$$\begin{aligned} \text{CPU clock time} &= 8 \times 0.24 + 7 \times 0.12 + 6 \times 0.44 + 5 \times 0.18 + 2 \times 0.02 \\ &= 6.3ns \end{aligned}$$

- Performance ratio is given by

$$\begin{aligned} \frac{\text{CPU performance}_{\text{variable clock}}}{\text{CPU performance}_{\text{single clock}}} &= \frac{\text{CPU execution time}_{\text{single clock}}}{\text{CPU execution time}_{\text{variable clock}}} \\ &= \frac{\text{IC} \times \text{CPU clock cycle}_{\text{single clock}}}{\text{IC} \times \text{CPU clock cycle}_{\text{variable clock}}} \\ &= \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}} \\ &= \frac{8}{6.3} \\ &= 1.27 \end{aligned}$$

- Example: Performance of single cycle CPU with floating point instructions
  - Left as reading assignment

A multicycle implementation

- The series of steps for each instruction can be used to create a multicycle implementation
  - Each step will take one clock cycle
  - Functional units can be used for more than one step per instruction, using different clock cycles
  - Figure 5.30 – Abstract version of multicycle datapath
    - A single memory unit for both instructions and data
    - Single ALU
    - Extra state elements to hold values from one unit to be used in a subsequent cycle by another unit
  - Additional registers or state elements
    - May not be visible to programmer
    - Placement determined by two factors
      1. Combinational units to fit in a clock cycle
      2. Data needed for later cycles in the instruction
  - A clock cycle can accomodate at most one of the following:
    - Memory access
    - Register file access (two reads or one write)
    - An ALU operation
  - Data produced by any of the above three operations must be saved in a state element for usage in subsequent operations
  - New registers added (all hold temporary values)
    - Instruction register (IR) and memory data register (MDR)
      - Hold the output from memory unit for instruction or data
      - Two separate registers because both values may be needed during the same clock cycle
    - Registers A and B

        - Hold the output of register file
    - ALUOut register
        - Holds the output of ALU
        - Required because the output of ALU is used in load/store instructions
- All registers except IR hold data between a pair of adjacent clock cycles, and do not need a write signal
    - IR needs to hold the instruction until end of execution of that instruction and so, needs a write signal
- Need new multiplexors, and expand existing ones, because of multiple uses of functional units
    - Multiplexor to distinguish between address coming from PC or ALUOut
- Replace three different ALUs in single cycle implementation with one
    - Additional multiplexor for first ALU input to choose between register A and PC
    - Multiplexor on second ALU input changed from 2 way to 4 way
        - Additional input constant 4 to increment PC
        - Sign-extended and shifted offset field for branch address computation
    - Figure 5.31
        - Added multiplexors and temporary registers but reduced multiple components such as memory and adders
        - Different control signals requirement due to multiple clock cycles per instruction
        - Need write signal for units visible to programmer (PC, memory, registers) and for IR
        - Need a read signal for memory
        - ALU control signal from single cycle datapath is used without any change
        - Each of 2-input multiplexors require 1 control input while 4-input multiplexors require 2 control inputs
        - Result in Figure 5.32
- Adding branches and jumps
    - Three possible sources for values to be written into PC
        1. Output of ALU – PC + 4 during instruction fetch
        2. Register `ALUOut` – Address of branch target when computed
        3. Lower 26 bits of IR shifted left by 2, with first 4 bits in 32-bit word coming from incremented PC
    - Write control signals for PC
        - Figure 5.33
        - Unconditional write achieved by `PCWrite` for normal increment or jump
        - Conditional write achieved by `PCWriteCond` when two designated registers are equal
          AND the zero signal of ALU with `PCWriteCond`
          Output of AND is OR'ed with unconditional `PCWrite`

- Breaking the instruction execution into clock cycles
    - Balance the work done in each clock cycle to minimize clock cycle time
    - Break each instruction into steps of one clock cycle each (ALU operation, register file access, or memory access)
    - Write data value in a register (user visible or internal) at the end of clock cycle for use in subsequent clock cycle
    - Edge-triggered design ensures that the register will not be modified until the next clock cycle and current value can be read until then
    - Some of the parallel steps in single cycle execution will need to be changed to series due to the fact that components like ALU need to be reused for different things (ALU operation plus incrementing PC)
    - Reading PC or standalone register is part of the original clock cycle while reading from register file requires additional clock cycle

    * Register file has additional controland access overhead
- Potential execution steps
  1. Instruction fetch step (Fig 5.33)

     ```
     IR = Memory[PC];
     PC += 4;
     ```

     * Send PC to memory, perform a read, and store the instruction into IR
       · Assert control signals `MemRead` and `IRWrite`
       · Set `IorD` to 0 to select PC as address source
     * Increment PC by 4
       · Set `ALUSrcA` to 0 (select PC)
       · Set `ALUSrcB` to 1 (select constant 4)
       · Set `ALUOp` to 00 (for add)
     * Store incremented address back into PC
       · Set `PCSource` to 00
       · Set `PCWrite`
     * Increment of PC and instruction memory access can occur in parallel
       · New value of PC not visible until next cycle
       · Incremented PC also stored in `ALUOut`
  2. Decode instruction and fetch registers (operands)
     * Use regularity of instruction format to optimistically perform some operations
     * Read registers indicated by `rs` and `rt`, even if it is not needed, storing read values in temporary registers A and B

       ```
       A = Reg[IR[25-21]];
       B = Reg[IR[20-16]];
       ```

     * Compute branch target address using ALU, saving the potential target in `ALUOut` (do it whether the instruction is branch or not)

       ```
       ALUOut = PC + ( sign_extend ( IR[15-0] ) << 2 );
       ```

  3. Execution, memory address computation, or branch completion
     (a) Memory instructions (computing the address for load/store)
       * Add operands to form a memory address

         ```
         ALUOut = A + sign_extend ( IR[15-0] );
         ```

       * Set `ALUSrcA` to 1
       * Set `ALUSrcB` to 10
       * Set `ALUOp` to 00 (for add)

     (b) *R*-type instructions
       * Perform operation on values read from register file

         ```
         ALUOut = A op B;
         ```

       * Set `ALUSrcA` to 1
       * Set `ALUSrcB` to 00
       * Set `ALUOp` to 10 to use `funct` field to determine ALU control signal settings

     (c) Branch instructions
       * Perform equality comparison between registers; use the result to determine whether to branch

         ```
         if ( A == B ) PC = ALUOut;
         ```

       * Set `ALUSrcA` to 1
       * Set `ALUSrcB` to 00
       * Set `ALUOp` to 01 to subtract
       * `PCWriteCond` needs to be asserted if `Zero` output of ALU is asserted

        * Set `PCSource` to 01
        * If `PCWriteCond` is asserted, the value in PC that was written in the instruction fetch step is overwritten and used for subsequenct instructions
    (d) Jump instruction
        * Replace PC by jump address

          ```
          PC = PC[31-28] || ( IR[25-0] << 2 );
          ```
        * Set `PCSource` to direct the jump address to PC
        * Assert `PCWrite` to write the jump address
4. Memory access or *R*-type instruction completion
    (a) Memory instructions
        * Load or store instruction

          ```
          MDR = Memory[ALUOut];
          Memory[ALUOut] = B;
          ```
        * Use the address stored in `ALUOut` as computed in previous step
        * Assert `MemRead` for load and `MemWrite` for store
        * Set `IorD` to 1 to force the memory address to come from ALU and not from PC
        * `MDR` is written in every clock cycle and so, no explicit clock signal need be asserted
    (b) *R*-type instructions
        * Write the result of ALU operation into register file

          ```
          Reg[IR[15-11]] = ALUOut;
          ```
        * Set `RegDst` to 1 to force the `rd` field to be used to select register entry
        * Assert `RegWrite`
        * Set `MemtoReg` to 0 to write the output of ALU rather than `MDR`
5. Memory read completion
    * Write the value from memory into register

      ```
      Reg[IR[20-16]] = MDR;
      ```
    * Set `MemtoReg` to 1 to write the result from `MDR`
    * Assert `RegWrite`
    * Set `RegDst` to 0 to use the `rt` field as register number

- Defining the control
  - In single cycle implementation, we used a set of truth tables to specify the setting of control signals
  - Now, control is more complex because of multiple steps in each instruction
  - Control must specify the signals to be set in current step as well as next step
  - Use the principles of finite state machines to specify control graphically
    * Allows detailed implementation using gates, ROMs, or PLAs to be synthesized by a CAD system
  - Finite state machine
    * Set of states and directions on how to change states
    * *Next-state function* to map from current state and inputs to a new state
    * Set of outputs asserted when the machine is in a state
    * Assumption: All outputs that are not asserted are de-asserted
    * Correct operation of datapath depends that a signal that is not asserted is actually de-asserted rather than don't care
  - Multiplexor controls
    * Select one of the inputs whether 0 or 1
    * Always specify the setting of all multiplexor controls needed

- Finite state control
  - Corresponds to the five steps of execution
  - Each state will take 1 clock cycle
  - Since first two steps are common in all instructions, initial two states will be common as well
  - The next three steps depend on the opcode, and will differ depending on instruction opcode
  - At the end of last step, the finite state machine starts over again
  - Figure 5.36 – overall picture
    - Each box corresponds to a subsystem that needs to be filled
  - Figure 5.37 – Fetch and decode
    - Number the states corresponding to steps in the automaton
    - State 0, or step 1, is the start state of the machine
    - Signals asserted in the state are shown within the circle representing the state
    - Four arcs leaving state 1, corresponding to four instruction classes – memory reference, *R*-type, branch on equal, and jump – labeled appropriately
    - Selection of path based on instruction decoding process

      Choice of next state depends on instruction class
  - Figure 5.38 – Memory reference
    - State to compute memory address by setting ALU input multiplexors for first input to register *A*
    - Second input is sign-extended displacement field
    - Result goes into ALUOut register
    - After address calculation, memory should be read (state 3) or written into (state 4)
    - Output of memory is always written into MDR
    - Signal `IoD` must be set to 1 to force the address to come from ALU instead of PC
    - The completion of instruction leads to the next state which is state 0
  - Figure 5.39 – *R*-type instruction
    - Step 3 (state 6) is to execute, or perform computation
    - Step 4 (state 7) is to complete instruction, or store result into register file
  - Figure 5.40 – Branch instruction
  - Figure 5.41 – Jump
  - Figure 5.42 – Complete finite state control machine