Basic functionalities of combinational and sequential circuits

Types of Logic Circuits: There are two types of Digital circuits depending on their output and memory used:

(i) Combinational circuit, and
(ii) Sequential circuit
A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs and they have no memory. A sequential circuit consists of logic gates whose outputs at any time are determined from both the present combination of inputs and previous output. That means sequential circuits use memory elements to store the value of previous output.

1. Combinational Circuits: These circuits are developed using AND, OR, NOT, NAND, and NOR logic gates. These logic gates are building blocks of combinational circuits. A combinational circuit consists of input variables and output variables. Since these circuits are not dependent upon previous input to generate any output, so are combinational logic circuits. A combinational circuit can have an n number of inputs and m number of outputs. In combinational circuits, the output at any time is a direct function of the applied external inputs.

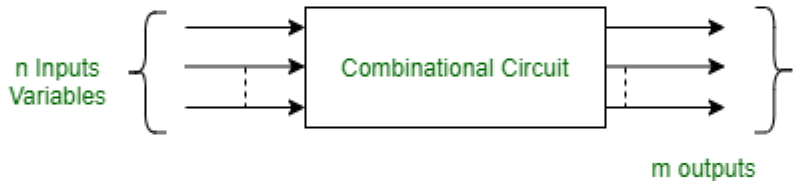


Figure - Block diagram of Combinational circuit

Design procedure of a Combinational Circuit
The design procedure of a combinational circuit involves the following steps:

- The problem is stated.
- The total number of available input variables and required output variables is determined.
- The input and output variables are allocated with letter symbols.
- The exact truth table that defines the required relationships between inputs and outputs is derived.
- The simplified Boolean function is obtained from each output.
- The logic diagram is drawn.
- The combinational circuit that performs the addition of two bits is called a half adder and the one that performs the addition of three bits (two significant bits and a previous carry) is a full adder.

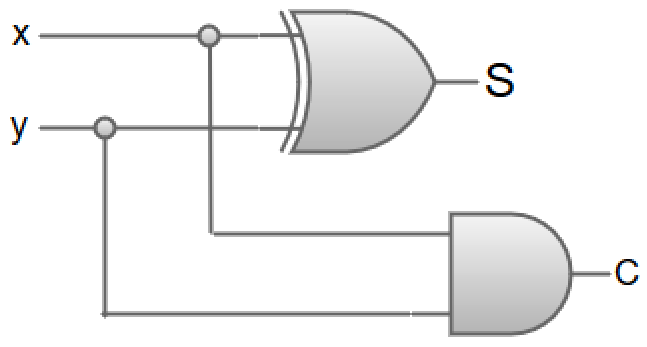
Half - Adder
A Half-adder circuit needs two binary inputs and two binary outputs. The input variable shows the augend and addend bits whereas the output variable produces the sum and carry. We can understand the function of a half-adder by formulating a truth table. The truth table for a half-adder is:

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

'x' and 'y' are the two inputs, and S (Sum) and C (Carry) are the two outputs.
The Carry output is '0' unless both the inputs are 1.
'S' represents the least significant bit of the sum.
The simplified sum of products (SOP) expressions is:

$S = x'y+xy'$, $C = xy$

The logic diagram for a half-adder circuit can be represented as:



$$S = x'y + xy'$$
$$C = xy$$

2. **Sequential circuits**: A sequential circuit is specified by a time sequence of inputs, outputs, and internal states. The output of a sequential circuit depends not only on the combination of present inputs but also on the previous outputs. Unlike combinational circuits, sequential circuits include memory elements with combinational circuits. Some examples are counters and shift registers.

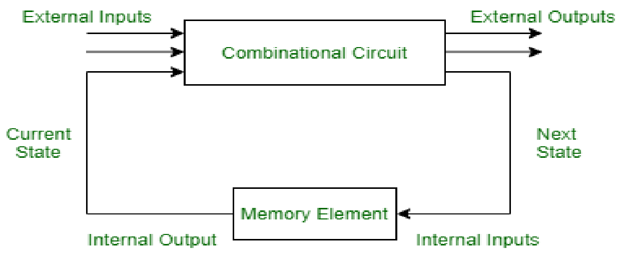


Figure - Sequential Circuit

- The memory elements are circuits capable of storing binary information.
- The binary information stored in these memory elements at any given time defines the state of the sequential circuit at that time.
- The external output of a sequential circuit depends both on the present input and the previous output state.
- The next state of the memory elements also depends on the external input and the present state of the external output.
- Some sequential circuits may not contain combinational circuits, but only memory elements.
- Generally, there are two types of storage elements used: Latches, and Flip-Flops. Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches; those controlled by a clock transition are flip-flops.

Difference between combinational and sequential circuit

Combinational circuits are defined as the time independent circuits which do not depends upon previous inputs to generate any output are termed as combinational circuits. Sequential circuits are those which are dependent on clock cycles and depends on present as well as past inputs to generate any output.

Combinational Circuit –

- In this output depends only upon present input.
- Speed is fast.
- It is designed easy.
- There is no feedback between input and output.
- This is time independent.
- Elementary building blocks: Logic gates
- Used for arithmetic as well as boolean operations.
- Combinational circuits don’t have capability to store any state.
- As combinational circuits don’t have clock, they don’t require triggering.
- These circuits do not have any memory element.
- It is easy to use and handle.
- Examples – Encoder, Decoder, Multiplexer, Demultiplexer

Sequential Circuit –

- In this output depends upon present as well as past input.
- Speed is slow.
- It is designed tough as compared to combinational circuits.
- There exists a feedback path between input and output.
- This is time dependent.
- Elementary building blocks: Flip-flops
- Mainly used for storing data.
- Sequential circuits have capability to store any state or to retain earlier state.
- As sequential circuits are clock dependent they need triggering.
- These circuits have memory element.
- It is not easy to use and handle.
- Examples – Flip-flops, Counters

S-R Flip-flop/Basic Flip-Flop

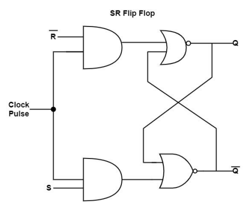
Flip flops are an application of logic gates. A flip-flop circuit can remain in a binary state indefinitely (as long as power is delivered to the circuit) until directed by an input signal to switch states.

S-R flip-flop stands for SET-RESET flip-flops.

The SET-RESET flip-flop consists of two NOR gates and also two NAND gates.

These flip-flops are also called S-R Latch.

The design of these flip flops also includes two inputs, called the SET [S] and RESET [R]. There are also two outputs, Q and Q'**.**



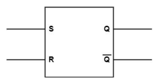
The truth table of SR flip flop is shown in the table.

S	**R**	**QN.1**	**$\overline{QN \cdot 1}$**
0	**0**	**QN**	**$\overline{QN}$**
0	**1**	**0**	**1**
1	**0**	**1**	**0**
1	**1**	**Indeterminate**	**Indeterminate**

A team of cross-coupled NOR gates can describe an SR flip-flop, wherein, the output of one gate is related to one of the two inputs of the other gate and vice versa. The complementary input of one NOR gate is ‘R’ while the complementary input of the other gate is ‘S’.

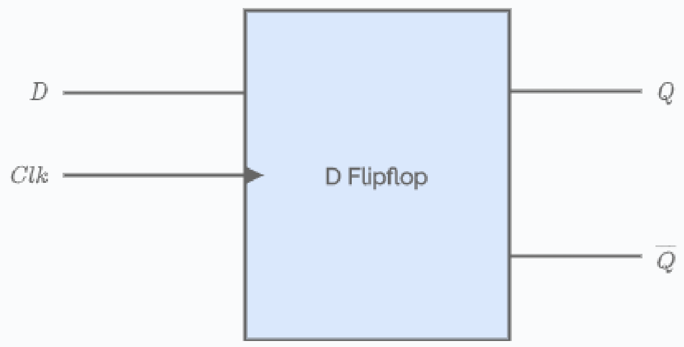
The input ‘R’ makes the output Q and the gate with the ‘S’ input makes the output $\overline{Q}$.

The logic symbol of the SR flip-flop is shown in the figure.



D Flip Flop

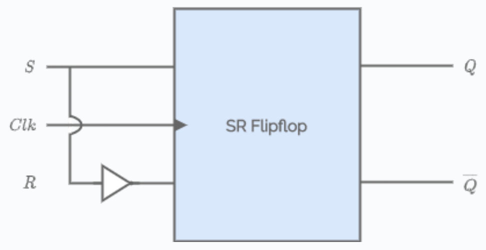
The D(data) flip-flop stores the value that is given on the data line. It can be thought of as a basic memory cell.



D Flip flop Block Diagram

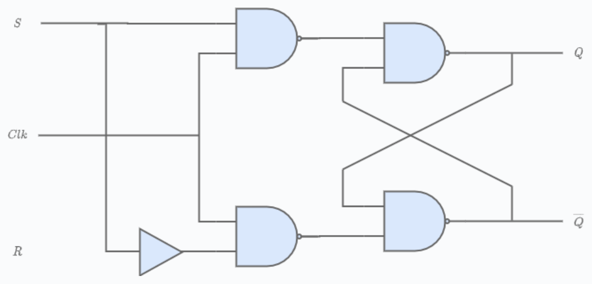
When both the inputs of an SR flip-flop are at the same logic level, then a no change or invalid output condition occurs. If we avoid these conditions, there will be only two output conditions: SET or RESET. There are many applications, where only SET and RESET conditions of the latch are required. In these applications, we can use inputs (S and R) which are always the complement of each other.

This can be designed by a single input (S) to the latch and the R input achieved by inverting this S. This single input is called data input and it is labelled with D.



Working of D Flip flop

The circuit diagram of a D flip-flop is given below:



When the clock is set to low, the output remains as it is whether the input signal is set high or low.

When clock is high, if D = 1 then it is equivalent to S = 1 and R = 0 hence the latch is set(1). On the other hand, when D = 0 then it acts like S = 0, and R = 1 hence the latch is reset(0). The characteristics table and truth table of a D flip flop is given below. clk= clock pin.

Q_t	D	Q_{t+1}
0	0	0
0	1	1
1	0	0
1	1	1

Characteristics table

Clk	D	Q_{t+1}
0	X	Qt
1	0	0
1	1	1

Truth table

What is Datapath?

The component of the processor that performs arithmetic operations – P&H

The collection of state elements, computation elements, and interconnections that together provide a conduit for the flow and transformation of data in the processor during execution.

Datapath Design and Implementation

The general discipline for datapath design is to (1) determine the instruction classes and formats in the ISA, (2) design datapath components and interconnections for each instruction class or format, and (3) compose the datapath segments designed in Step 2) to yield a composite datapath.

Simple datapath components include *memory* (stores the current instruction), *PC* or program counter (stores the address of current instruction), and *ALU* (executes current instruction). The interconnection of these simple components to form a basic datapath is illustrated in Figure 4.5. Note that the register file is written to by the output of the ALU. The register file shown in Figure is clocked by the RegWrite signal.

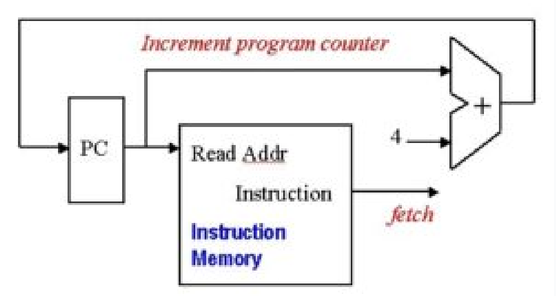


Figure 1. Schematic high-level diagram of MIPS datapath from an implementational perspective

Implementation of the datapath for I- and J-format instructions requires two more components – a *data memory* and a *sign extender*, illustrated in Figure 2. The data memory stores ALU results and operands, including instructions, and has two enabling inputs (MemWrite and MemRead) that cannot both be active (have a logical high value) at the same time. The data memory accepts an address and either accepts data (WriteData port if MemWrite is enabled) or outputs data (ReadData port if MemRead is enabled), at the indicated address. The sign extender adds 16 leading digits to a 16-bit word with most significant bit *b*, to product a 32-bit word. In particular, the additional 16 digits have the same value as *b*, thus implementing sign extension in twos complement representation.

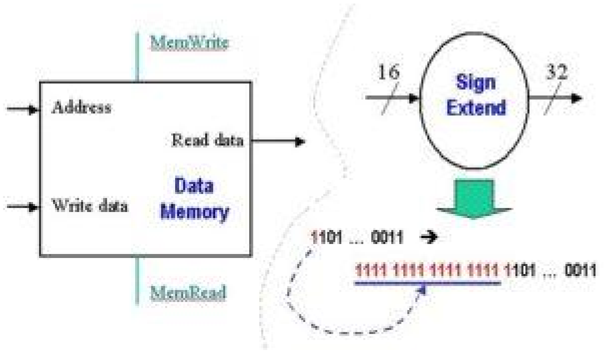


Figure 2. Schematic diagram of Data Memory and Sign Extender

R-format Datapath

Implementation of the datapath for R-format instructions is fairly straightforward – the register file and the ALU are all that is required. The ALU accepts its input from the DataRead ports of the register file, and the register file is written to by the ALUresult output of the ALU, in combination with the RegWrite signal.

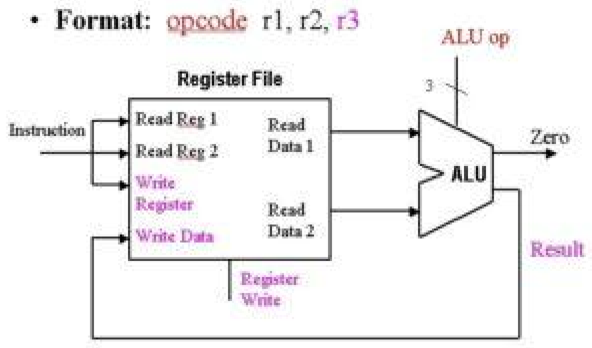
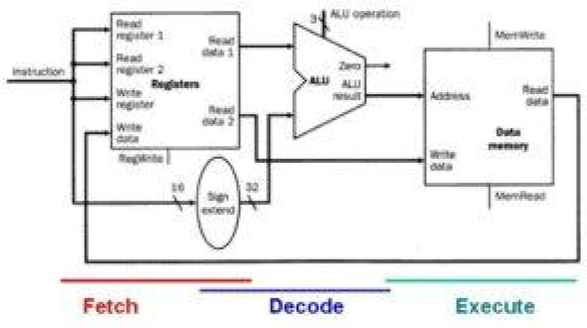


Figure 3. Schematic diagram R-format instruction datapath

Load/Store Datapath

The load/store datapath uses instructions such as lw $t1, **offset**($t2), where *offset* denotes a memory address offset applied to the base address in register $t2. The lw instruction reads from memory and writes into register $t1. The sw instruction reads from register $t1 and writes into memory. In order to compute the memory address, the MIPS ISA specification says that we have to sign-extend the 16-bit offset to a 32-bit signed value. This is done using the sign extender shown in Figure 2.
The load/store datapath is illustrated in Figure 4, and performs the following actions in the order given:

1. ***Register Access*** takes input from the register file, to implement the instruction, data, or address *fetch* step of the fetch-decode-execute cycle.
2. ***Memory Address Calculation*** decodes the base address and offset, combining them to produce the actual memory address. This step uses the sign extender and ALU.
3. ***Read/Write from Memory*** takes data or instructions from the data memory, and implements the first part of the *execute* step of the fetch/decode/execute cycle.
4. ***Write into Register File*** puts data or instructions into the data memory, implementing the second part of the *execute* step of the fetch/decode/execute cycle.



Branch/Jump Datapath

The branch datapath (jump is an unconditional branch) uses instructions such as beq $t1, $t2, **offset**, where *offset* is a 16-bit offset for computing the branch target address via PC-relative addressing. The beq instruction reads from registers $t1 and $t2, then compares the data obtained from these registers to see if they are equal. If equal, the branch is taken. Otherwise, the branch is not taken.
By *taking the branch*, the ISA specification means that the ALU adds a sign-extended offset to the program counter (PC). The offset is shifted left 2 bits to allow for word alignment (since 22 = 4, and words are comprised of 4 bytes). Thus, to jump to the target address, the lower 26 bits of the PC are replaced with the lower 26 bits of the instruction shifted left 2 bits.
The branch instruction datapath is illustrated in Figure 4.9, and performs the following actions in the order given:

1. ***Register Access*** takes input from the register file, to implement the *instruction fetch* or *data fetch* step of the fetch-decode-execute cycle.
2. ***Calculate Branch Target*** – Concurrent with ALU #1's evaluation of the branch condition, ALU #2 calculates the branch target address, to be ready for the branch if it is taken. This completes the *decode* step of the fetch-decode-execute cycle.
3. ***Evaluate Branch Condition and Jump*** *to BTA or PC+4* uses ALU #1 in Figure 5, to determine whether or not the branch should be taken. Jump to BTA or PC+4 uses control logic hardware to transfer control to the instruction referenced by the branch target address. This effectively changes the PC to the branch target address, and completes the *execute* step of the fetch-decode-execute cycle.

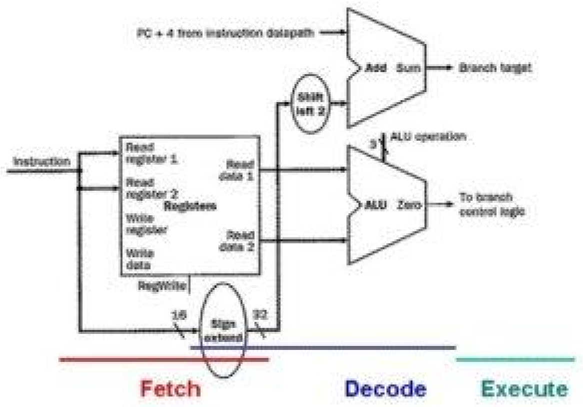


Figure 5. Schematic diagram of the Branch instruction datapath. Note that, unlike the Load/Store datapath, the *execute* step does not include writing of results back to the register file [MK98].
The branch datapath takes operand #1 (the offset) from the instruction input to the register file, then sign-extends the offset. The sign-extended offset and the program counter (incremented by 4 bytes to reference the next instruction after the branch instruction) are combined by ALU #1 to yield the branch target address. The operands for the branch condition to evaluate are concurrently obtained from the register file via the ReadData ports, and are input to ALU #2, which outputs a one or zero value to the branch control logic.

Single-Cycle Datapaths

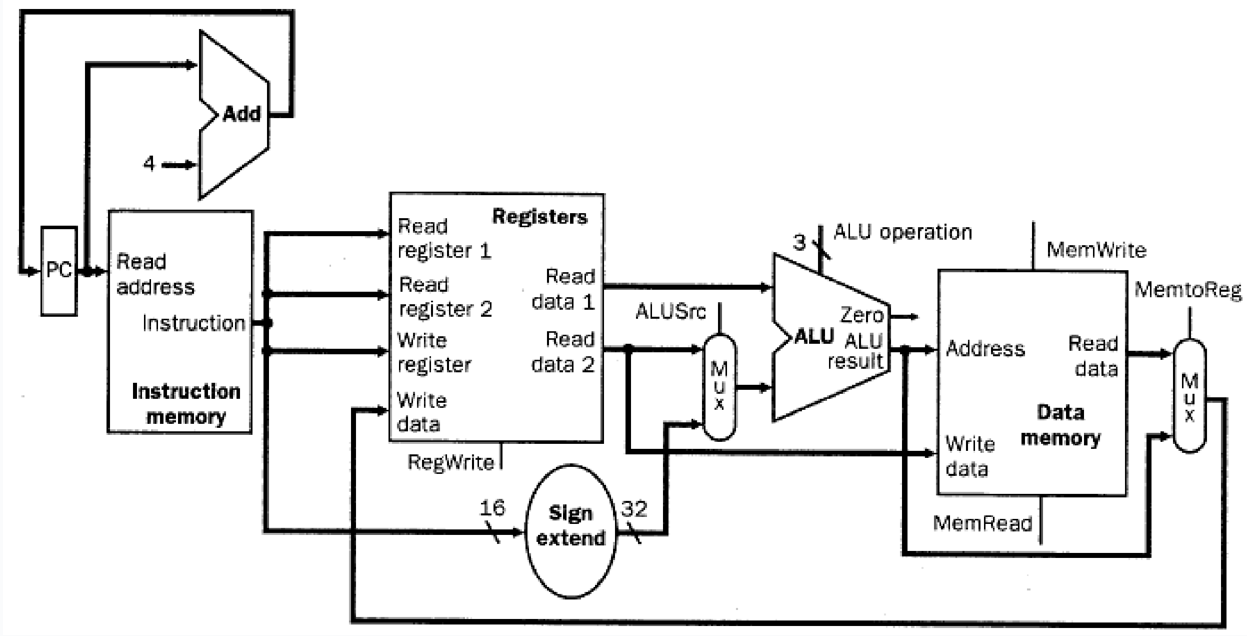
A single-cycle datapath executes in one cycle all instructions that the datapath is designed to implement. This clearly impacts CPI in a beneficial way, namely, CPI = 1 cycle for all instructions.

The key to efficient single-cycle datapath design is to find commonalities among instruction types. For example, the R-format MIPS instruction datapath of Figure 3 and the load/store datapath of Figure 4 have similar register file and ALU connections. However, the following differences can also be observed:

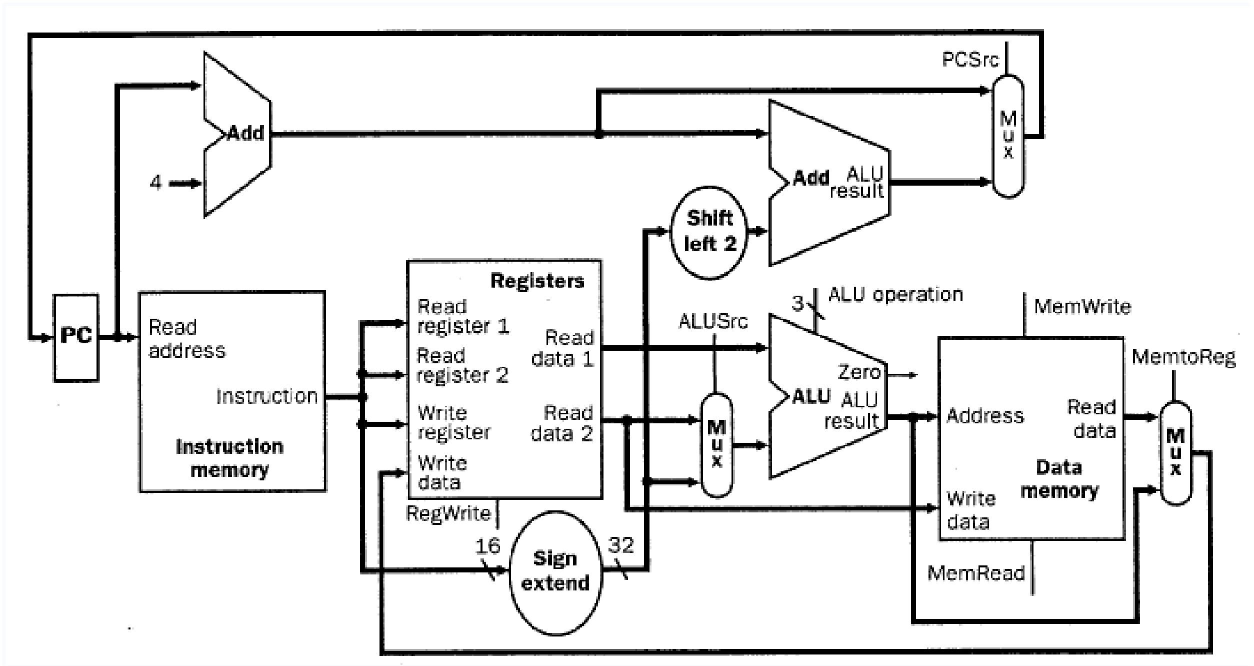
The second ALU input is a register (R-format instruction) or a signed-extended lower 16 bits of the instruction (e.g., a load/store offset).

The value written to the register file is obtained from the ALU (R-format instruction) or memory (load/store instruction).

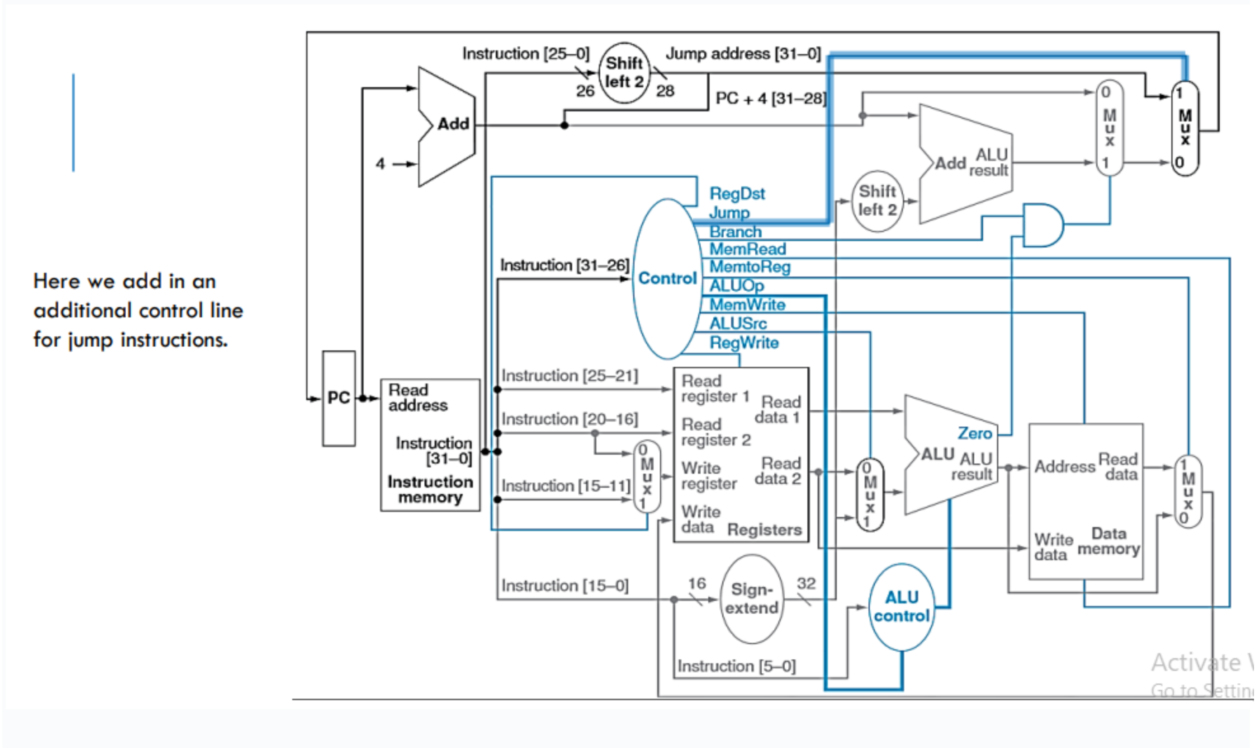
These two datapath designs can be combined to include separate instruction and data memory, as shown in Figure 6. The combination requires an adder and an ALU to respectively increment the PC and execute the R-format instruction.



Adding the branch datapath to the datapath illustrated in Figure 2 produces the augmented datapath shown in Figure 1. The branch instruction uses the main ALU to compare its operands and the adder computes the branch target address. Another multiplexer is required to select either the next instruction address (PC + 4) or the branch target address to be the new value for the PC.



ALU Control. Given the simple datapath shown in Figure 2, we next add the control unit. Control accepts inputs (called control signals) and generates (a) a write signal for each state element, (b) the control signals for each multiplexer, and (c) the ALU control signal.


Here we add in an additional control line for jump instructions.

Drawbacks of single cycle implementation

• All instructions take the same time although – some instructions are longer than others;

• e.g. load is longer than add since it has to access data memory in addition to all the other steps that add does

• Some combinational units must be replicated since used in the same cycle

– e.g., ALU for computing branch address and ALU for computing branch outcome

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 in Section 4.3.1 involves the following steps:

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

Note that this implementational sequence is actually combinational, becuase of the single-cycle assumption. Since the datapath operates within one clock cycle, the signals stabilize approximately in the order shown in Steps 1-4, above.

Load/Store Instruction Execution of a load/store instruction (e.g., lw $t1, offset($t2)) using the datapath developed in previous section involves the following steps:

- Fetch instruction from instruction memory and increment PC
- Read register value (e.g., base address in $t2) from the register file
- ALU adds the base address from register $t2 to the sign-extended lower 16 bits of the instruction (i.e., offset)
- Result from ALU is applied as an address to the data memory
- Data retrieved from the memory unit is written into the register file, where the register index is given by $t1 (Bits 20-16 of the instruction).

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

- Fetch instruction from instruction memory and increment PC
- 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).
- 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.

Drawbacks of single cycle implementation

• All instructions take the same time although
 – some instructions are longer than others;
  • e.g. load is longer than add since it has to access data memory in addition to all the other steps that add does

• Some combinational units must be replicated since used in the same cycle
 – e.g., ALU for computing branch address and ALU for computing branch outcome

Pipeline Hazard

Earlier we had mentioned that the memory limits the speed of the CPU. Now there is one more case. In a pipelined design few instructions are in some stage of execution. There are possibilities for some kind of dependency amongst these set of instructions and thereby limiting the speed of the Pipeline.

We made the following observations about pipelining:

- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage o Multiple tasks operating simultaneously
- Potential speedup = Number of pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup o Unbalanced lengths of pipe stages reduces speedup
- Execute billions of instructions, so throughput is what matters o Data path design – Ideally we expect a CPI value of 1

There are three kinds of hazards:

- **Structural hazards**: Hardware cannot support certain combinations of instructions (two instructions in the pipeline require the same resource).
- **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
- **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

Structural hazards arise because there is not enough duplication of resources.

Resolving structural hazards:

Solution 1: Wait

o Must detect the hazard
o Must have mechanism to stall
o Low cost and simple
o Increases CPI
o Used for rare cases

Solution 2: Throw more hardware at the problem

o Pipeline hardware resource

§ useful for multi-cycle resources
§ good performance
§ sometimes complex e.g., RAM

o Replicate resource

§ good performance
§ increases cost (+ maybe interconnect delay)

§ useful for cheap or divisible resource

Data Hazards classification

Data hazards are classified into three categories based on the order of READ or WRITE operation on the register and as follows:

1. **RAW (Read after Write) [Flow/True data dependency]**
   This is a case where an instruction uses data produced by a previous one. Example

   ADD R0, R1, R2
   SUB R4, R3, R0

2.WAR (Write after Read) [Anti-Data dependency]
This is a case where the second instruction writes onto register before the first instruction reads. This is rare in a simple pipeline structure. However, in some machines with complex and special instructions case, WAR can happen.

ADD R2, R1, R0
SUB R0, R3, R4

3.WAW (Write after Write) [Output data dependency]
This is a case where two parallel instructions write the same register and must do it in the order in which they were issued.

ADD R0, R1, R2
SUB R0, R4, R5

WAW and WAR hazards can only occur when instructions are executed in parallel or out of order. These occur because the same register numbers have been allotted by the compiler although avoidable. This situation is fixed by renaming one of the registers by the compiler or by delaying the updating of a register until the appropriate value has been produced

What are the hazards that can happen in our simple MIPS pipeline?

- Structural hazard
- Conflict for use of a resource
- • In MIPS pipeline with a single memory
  - – Load/store requires data access
  - – Instruction fetch would have to stall for that cycle
-
- • Would cause a pipeline “bubble”
- • Hence, pipelined datapaths require separate instruction/data memories or separate instruction/data caches
- RAW hazards – can happen in any architecture
- WAR hazards – Can’t happen in MIPS 5 stage pipeline because all instructions take 5 stages, and reads are always in stage 2, and writes are always in stage 5
- WAW hazards – Can’t happen in MIPS 5 stage pipeline because all instructions take 5 stages, and writes are always in stage 5
- Control hazards