The single cycle implementation of the MIPS datapath was inefficient.
  - Clock cycle time is dictated by the longest instruction. The rest of the instructions are wasting time doing nothing but waiting for the clock.
  - Throughput (number of instructions per unit time) as a result is low

One approach to address this issue is to split the datapath into smaller pieces/stages each which can be performed efficiently in a single clock cycle.
  - Each type of instruction must then execute only the stages of the datapath that are required for the instruction.
  - The clock cycle time is now dictated by the longest stage of the datapath rather than the full instruction.
  - However, now extra control is needed to determine how many and which stages are required for the current instruction.
    - In addition to using the opcode to determine the control signals, now we need to know which stage of the datapath we are executing and which is next.
  - An additional advantage is that we can reuse components of the data path. Within each piece of the datapath a component can only be used once, but a unit can be used for different purposes during different pieces.
    - The ALU can be used to compute PC+4, the memory address, and the branch address
    - A single instruction/data memory can be used rather than separate units

The Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done in each cycle, meaning try to keep the timing of each piece similar and the tasks per cycle logical
  - restrict each cycle to use only one major functional unit (this functional unit is the bottleneck, 2 units would make the cycle time longer)

- At the end of a cycle, we must
  - store the intermediate values for use in the later cycles
  - We can do so by introducing additional “internal” registers

- Below is the multicycle datapath:
  - Red circles indicate the added registers.
  - The Instruction memory and data memory are now combined in to a single unit (purple).
The adders are removed. ALU can perform all necessary operations.
Additional multiplexors are added to select between more options (green). The Jump mux is removed.

The above datapath changes were decided by considering the MIPS instruction set/ISA

Ex: add $1, $2, $3
  - The Instruction was specified by the PC address in memory
  - The instruction changes a register, which is specified by bits 15:11 of instruction.
  - The value to store in the register is the sum (“op”) of two registers, specified by bits 25:21 and 20:16 of the instruction

  Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]] op Reg[Memory[PC][20:16]]

  We can break up the above statement into smaller manageable pieces like we would in HLL programming (introducing variables)
  - Could break down to:
    - IR <= Memory[PC]
    - A <= Reg[IR[25:21]]
    - B <= Reg[IR[20:16]]
    - ALUOut <= A op B
    - Reg[IR[15:11]] <= ALUOut

  We forgot an important part of the instruction!
  - PC <= PC + 4

  Each of the intermediate variables are where we have to store temporary variables/values. These are the new registers in the datapath.

Five Execution Stages of the Multicycle Datapath

- The multicycle datapath is split into 5 stages (one for each major task in the instruction, only one main functional unit per stage)
  - Instruction Fetch and PC <= PC + 4
  - Instruction Decode and Register Fetch
  - Execution, Memory Address Computation, or Branch Completion
  - Memory Access or R-type instruction completion
  - Write-back step

Each instruction will only use a subset of these stages. Therefore, instructions will take from 3-5 clock cycles to execute. As a result, each instruction will take different amounts of time to execute.

Stage 0: Instruction Fetch
- There are two main tasks
  - Use PC to retrieve the instruction from the Memory and store the 32-bit instruction value in the Instruction Register. IR <= Memory[PC]
  - Increment the PC by 4 and store the result back in the PC. PC <= PC + 4;
- This stage is instruction independent. No knowledge of the type of instruction is available.
**Stage 1: Instruction Decode and Register Fetch (Reg Read)**

- This stage has 3 components:
  - Use the opcode to determine which instruction this is (control unit)
  - Read registers rs and rt from the register file (in case we need them)
    - \( A \leftarrow \text{Reg}[IR[25:21]] \);
    - \( B \leftarrow \text{Reg}[IR[20:16]] \);
  - Compute the branch address (in case the instruction is a branch)
    - \( \text{ALUOut} \leftarrow \text{PC} + (\text{sign-extend}(\text{IR}[15:0]) \ll 2) \);
    
    The ALU was not being used for anything in this step, therefore we anticipate the need for the branch address.

- This stage is instruction independent. The control lines are not set on the instruction type yet (we are busy "decoding" it in our control logic during this stage)
- **Stage 2: ALU**
  - This stage is instruction dependent. ALU is performing one of 4 functions, based on instruction type (memory, R-type, Branch, jump). Note Jump is not illustrated below.
  - **Bubble 2:** Memory Reference/immediate instructions (lw, sw, addi, etc):
    \[
    \text{ALUOut} \leftarrow A + \text{sign-extend}(\text{IR}[15:0])
    \]

- **Stage 2: R-type** (add, sub, slt, etc):
  - **Bubble 6:** ALU operations
    \[
    \text{ALUOut} \leftarrow A \text{ op } B
    \]
- **Stage 2: Branch** \((\text{beq})\):
  - **Bubble 8**: comparison of values for branch
    
    \[
    \text{if } (A==B) \text{ PC } \leftarrow \text{ALUOut};
    \]

- **Stage 3**: Memory \((\text{lw})\)
  - This stage is only required for R-type or memory-access \((\text{lw,sw})\)
  - **Bubble 3**: Load Word
    
    \[
    \text{MDR } \leftarrow \text{Memory[ALUOut]};
    \]
- **Stage 3: Memory (lw)**
  - **Bubble 5:** Store Word
    
    \[ \text{Memory}[\text{ALUOut}] \leftarrow B; \]

- **Stage 3: Reg File (r-type)**
  - **Bubble 7:** R-type instructions store to the register rd
    
    \[ \text{Reg}[\text{IR}[15:11]] \leftarrow \text{ALUOut}; \]
• **Stage 5:** Write Back (WB)
  - This is instruction dependent and only required for load word instructions.
  - **Bubble 4:** Write the data from memory into the registers
    \[ \text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR}; \]

Multi-cycle Implementation Summary

- The chart below summarizes all of the different operations which occur in each stage/step.
- Remember each stage/step is a single clock cycle. The clock cycle time is dependent on the critical path for any of the different steps.
- Each instruction type (columns) require a different number of clock cycles to complete depending on the number of stages/steps required.
  - R-type: 4 clock cycles
  - Lw: 5 clock cycles
  - Sw: 4 clock cycles
  - Beq: 3 clock cycles
  - J: 3 clock cycles
- The length of the clock cycle is determined by the length of the longest critical path. The critical path must be calculated for each of these stages/steps separately.
  - Remember: for any clock cycle, the processor does not know which stage/step or the instruction it is executing. So the clock has to operate for the worst case timing.

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>IR &lt;= Memory[PC]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC &lt;= PC + 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td></td>
<td>A &lt;= Reg[IR[25:21]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B &lt;= Reg[IR[20:16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut &lt;= PC + (sign-extend(IR[15:0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut &lt;= A op B</td>
<td>ALUOut &lt;= A + sign-extend(IR[15:0])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg[IR[15:11]] &lt;= ALUOut</td>
<td>Load: MDR &lt;= Memory[ALUOut] or Store: Memory[ALUOut] &lt;= B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[IR[20:16]] &lt;= MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** IR is shown in the figure as “Instr”. MDR is shown in the figure as “Data”
We have now discussed the different stages for the basic MIPS instructions (lw, sw, add, and, or, slt, beq, j). Next we will discuss the control for the multi-cycle datapath.

**Some Simple Questions:**

Consider the following MIPS code:

```mips
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label  #assume not
add $t5, $t2, $t3
sw $t5, 8($t3)
```

Label: ...

How many cycles will it take to execute this code?

21 clock cycles = 5+5+3+4+4 = lw + lw + beq + R-type + sw

What is going on during the 8th cycle of execution?

lw + partial lw = 5 + 3 for second lw. The third stage of lw is ALU.
ALUOut = Reg[$t3] + sign extended immediate value 4

In what cycle does the actual addition of $t2 and $t3 takes place?

Cycle 16. lw + lw + beq + add until ALU stage = 5+5+3+3
MIPS Multicycle Datapath Control

- The multicycle datapath created smaller stages to perform the tasks for each instruction.
- For each of these stages, the control signals for the entire datapath must be determined.
- Because each stage will now operate as an independent clock cycle, we must document/remember which state we are currently in and determine which state to go to next for the particular type of instruction.
- This type of control is performed using a Finite State Machine.

Finite State Machines
- Finite state machines, have a finite set of defined states. The name of each state (or binary value) is used to determine both the next state and the output values.
  - The next state is determined by the current state and any input values
  - The output values are determined by the current state and possibly the input values

Ex: Electronic Eye
- A friend would like you to build an “electronic eye” for use as a fake security device.
  - The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on.
  - Only one light is on at a time, and the light “moves” from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity.
  - How do we represent the finite state machine used to specify the electronic eye?
    - Note that the rate of the eye’s movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.

- First, we can determine the transitions between the lights by creating 3 states, blink L, M, R and connecting them.
  - A problem arises when you are in BlinkM. How do you know which transition to take #2 or #4
  - To remedy this issue, we can split Blink M into 2 separate states
Now for each state there is a clear single transition.
Inside each state bubble we can specify which of the lights should be lit, aka the control signals for the lights when in this state.

**Multicycle Datapath Control**
- We use a finite state machine to denote the state transitions and the control values for each type of operation in the multicycle datapath.
- To implement the control in this case, the value of the datapath control signals (all outputs from the control Logic oval) are dependent upon:
  - what instruction is being executed
  - which step is being performed
- In the following figure we denote each of the unique states required for all basic instructions.
- **Note:**
  - If a control signal is not mentioned in the bubble, then it is a don’t care (except for write signals)
  - Any control that is deals with write (PCWrite, RegWrite, IRWrite, etc) is deasserted (0) if not specified
  - If the name is listed only, this means the value of the control is asserted (1)
- Note the two red values in the figure (these are the correct values) make sure to correct them in all other handouts/figures
- The current finite state machine has 10 states, therefore 4 pins are required to specify the state number (shown to the top left of each bubble)
- Transitions between states are determined by the opcode when specified.
- Note that state 0 and 1 are instruction independent and will always be executed for any instruction (basic and new).
- The control values in each bubble/state are specified by the datapath for each stage for each type of instruction as presented in last lecture.

Microprogramming
- An alternate approach to finite state machines is to use microprogramming
- A static program for each instruction is created. The processor executes one microinstruction per clock cycle.
- Similar to the finite state machine the microinstruction itself consists of several control signals for the datapath. If a particular microinstruction bit is 1, then the particular corresponding output is 1.
- The size of microinstruction is dictated by the number of control bits we need.
- For each MIPS instruction (eg. lw, sw, slt, beq), has its own microprogram.
  - Execution of an instruction means executing its microprogram.
- The microprograms for each instruction is stored in a ‘control store’ which is a ROM (read only memory)
- Why?
A microprogram never changes. The instructions and the sequence of these microinstructions is set in stone when the datapath is designed.

A given instruction is always executed the same way, therefore the control store is written when the processor chip is made. There is no need to rewrite it.

- **How do we get the first microinstruction?**
  - The opcode usually provides the address of the first microinstruction in the corresponding microprogram. This is done in the decode stages. The Fetch and decode microinstructions are the same in all microprograms.

- The microcode storage works like a look-up table.
- A microinstruction also carries some explicit information about the next microinstruction (address). This is the sequencing control part. Sometimes we add 1 to obtain the next address, but not always (think end of instruction).

- We also have a microprogram counter (MPC).
  - It stores the address of the next microinstruction.
  - Many (MIPS) instructions may share some microcode. Similar to how states in the finite state machine are shared.
  - Using explicit sequencing (+1 increment) control keeps the size of the ROM small. Less repeated code. Less computation to calculate next microinstruction.

### MIIPS Multicycle Datapath – Adding New Instructions

- Modifications to the Multicycle datapath must be considered in stages.
- Any new instructions should be broken into subtasks/substages which are similar to the existing stages.
- Whenever possible the existing datapath stages and control states should be reused.
- Modification to the datapath (new data lines, multiplexors, etc) should only be added when necessary.
- Large components, such as ALUs, Memories, Registers, etc should only be added as a last resort.
- Care must be made not to increase the critical path of any stage. In this case, adding an extra clock cycle is preferred over increasing the critical path. Why? Extra stages only affect that instruction type rather than all
Adding new instructions

- **Ex:** `addi $1, $2, immediate`
  - This instruction reads the $rs and $rt register.
  - The immediate value is added to the contents of Reg[rs] and stored into Reg[rt]
  - Of all the existing types of instructions, which type of instruction is this most like?
    - It is most like the lw/sw instructions which add Reg[rs] + immediate to calculate the memory address
    - Therefore we should consider using the stages/steps from lw/sw to implement addi
  - The instruction Fetch and Decode stage are always used for every instruction.
  - S 3, calculates the Reg[rs] + immediate value and places the result in ALUOut.
  - Next, we need to implement Reg[rt] = ALUOut. Always check if a state can be reused prior to creating a new one. In this case, none of the existing states perform this operation.
    - State 4, performs Reg[rt] = MDR.
    - State 7, performs Reg[rd] = ALUOut
  - A new state should be added to the figure to perform. No modification are required to the datapath to implement the instruction.
- **Ex: jr $v0**
  - Jump to the address specified by the $register
  - \# PC = Reg[rs]
  - This instruction can use either the R or I format as it only requires 1 register to be specified
  - There are multiple ways to add this instruction to the datapath and control. Depending on the goal of the system different choices can be made.
  - Below are 3 different working approaches to adding jr to the datapath.
  - **Approach #1 – modify datapath**
    - Fetch and decode are performed as normal.
    - At this point Reg[rs] is currently in the A register of the datapath.
    - Modification to the datapath can be made to move the contents of Reg[rs] to the PC. This would be the third stage.
      - The PCSource mux should be modified to take the output of the A register on input 3.
      - All existing stages do not use this new mux selection and therefore a new state must be created for this third stage.
Approach #2 – no datapath modification (but the long way)

- Fetch and decode are performed as normal.
- At this point Reg[rs] is currently in the A register of the datapath.
- We need to find a path for this value to the Register file. We can move the data to the register file through the ALU and the ALUOut register. This requires an additional clock cycle.
- In Stage 3 of the instruction we can pass A through the ALU by adding/subtracting with 0. Where does the 0 come from? In both the R and I instruction formats a second register can be specified. If we assume the assembler places the $0 register (00000) in the instructions field rt, then in the decode stage the value zero will be placed in register B. If we additionally assume the instruction format is R-type, we can use the function field to specify the subtraction for the ALUControl. In this way we can use the existing stage for R-type instructions.
- The result of the calculation is stored in ALUOut. In the next cycle, PC = ALUOut. This requires a new state.
- **Approach #3 – no datapath modification (the short way)**

  - Fetch and decode are performed as normal.
  - At this point Reg[rs] is currently in the A register of the datapath.
  - We need to find a path for this value to the Register file. We can move the data to the register file through the ALU and the ALUOut register. This requires an additional clock cycle.
  - In Stage 3 of the instruction we can pass A through the ALU by adding/subtracting with 0. Where does the 0 come from? In both the R and I instruction formats a second register can be specified. If we assume the assembler places the $0 register (00000) in the instructions field rt, then in the decode stage the value zero will be placed in register B.
  - The result of the calculation can be redirected to the PC using the existing path through PCSrcSource from the fetch cycle.
- **Ex**: `swinc $rt, immediate($rs)`
  - # `Mem[Reg[rs] + immediate] = Reg[rt]`
  - # `Reg[rs] = Reg[rs] + 4`
  - This instruction performs a standard sw, but additionally increments the value in rs by 4.
  - Based on this, we should be considering the stages and control flow for the sw instruction.
  - Fetch and decode are performed as normal.
  - The 3\textsuperscript{rd} stage (state 2) calculates the `Reg[rs] + immediate` values. The ALU is being used and therefore cannot be used to calculate `Reg[rs] + 4` in the same cycle.
  - The 4\textsuperscript{th} stage (state 5) does the write to memory, `Mem[ALUOut] = Reg[rt]`. During the cycle the ALU is not being used. Therefore we can use the time to perform `ALUOut = Reg[rs] + 4`. Note that these two actions both use ALUOut. Since values are not stored into the registers until the falling edge of the clock cycle, the value of `Reg[rs] + 4` will not OVERWRITE the memory address value until after the memory is completed its operation.

- A new stage now must be added to perform `Reg[rs] = ALUOut`. To do this, the RegDst mux must be modified to take the rs register as an input. Because of the modification to the datapath, this new stage, must have a new control state.