First, revisit the datapath for add, sub, lw, sw. We will augment it to accommodate the beq and j instructions.

**Execution of branch instructions**

```
beq $at, $zero, L
```
```
add $v1, $v0, $zero
add $v1, $v1, $v1
j somewhere
L: add $v1, $v0, $v0
```

The offset must be added to the next PC to generate the target address for branch.
The modified version of MIPS

We need a second adder, since the ALU is already doing subtraction for the beq.

- PCSrc=1 branches to PC+4+(offset×4)
- PCSrc=0 continues to PC+4.

The final datapath for single cycle MIPS. Find out which paths the signal follow for lw, sw, add and beq instructions
Executing R-type instructions

The **ALUop** will be determined by the value of the **opcode** field and the **function** field of the instruction word.
Executing LW instruction
Executing beq instruction

The branch may or may not be taken, depending on the ALU’s Zero output.
Control signal table

This table summarizes what control signals are needed to execute an instruction. The set of control signals vary from one instruction to another.

<table>
<thead>
<tr>
<th>Operation</th>
<th>RegDst</th>
<th>RegWrite</th>
<th>ALUSrc</th>
<th>ALUOp</th>
<th>MemWrite</th>
<th>MemRead</th>
<th>MemToReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>or</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>slt</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>010</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

How to implement the control unit? Recall how to convert a truth table into a logical circuit! This is implemented by the control unit.
The Control Unit

All control signals are not shown here
1-cycle implementation is not used
Why? Because the length of the clock cycle will always be determined by the slowest operation (lw, sw) even when the data memory is not used.

Practical implementations use multiple cycles per instruction, which fixes some shortcomings of the 1-cycle implementation.

- Faster instructions (R-type) are not held back by the slower instructions (lw, sw)
- The clock cycle time can be decreased, i.e. faster clock can be used
- Eventually simplifies the implementation of pipelining, the universal speed-up technique.

This requires some changes in the datapath
Multi-cycle implementation of MIPS

First, revisit the 1-cycle version

- PCSrc=1 branches to PC+4+(offset×4)
- PCSrc=0 continues to PC+4.

We need a second adder, since the ALU is already doing subtraction for the beq.
The multi-cycle version

Note that we have **eliminated two adders**, and used only **one memory unit** (so it is Princeton architecture) that contains both instructions and data. It is **not essential to have a single memory** unit, but it shows an alternative design of the datapath.
Intermediate registers are necessary

In each cycle, a fraction of the instruction is executed

Five stages of instruction execution

Cycle 1. Instruction fetch and PC increment
Cycle 2. Reading sources from the register file
Cycle 3. Performing an ALU computation
Cycle 4. Reading or writing (data) memory
Cycle 5. Storing data back to the register file
**Why intermediate registers?**

Sometimes we need the output of a functional unit in a later clock cycle during the execution of an instruction.

(Example: The instruction word fetched in stage 1 determines the destination of the register write in stage 5. The ALU result for an address computation in stage 3 is needed as the memory address for `lw` or `sw` in stage 4.)

**These outputs must be stored in intermediate registers for future use.** Otherwise they will be lost after the next clock cycle.

(Instruction read in stage 1 is saved in Instruction register. Register file outputs from stage 2 are saved in registers A and B. The ALU output will be stored in a register ALUout. Any data fetched from memory in stage 4 is kept in the Memory data register MDR.)
The Five Cycles of MIPS

(Instruction Fetch)
IR:= Memory[PC]
PC:= PC+4

(Instruction decode and Register fetch)
A:= Reg[IR[25:21]], B:= Reg[IR[20:16]]
ALUout := PC + sign-extend(IR[15:0])

(Execute|Memory address|Branch completion)
Memory reference: ALUout:= A+ IR[15:0]
R-type (ALU): ALUout:= A op B
Branch: if A=B then PC := ALUout

(Memory access | R-type completion)
LW: MDR:= Memory[ALUout]
SW: Memory[ALUout]:= B
R-type: Reg[IR[15:11]]:= ALUout

(Writeback)
LW: Reg[[20:16]]:= MDR
Instruction execution review

- Executing a MIPS instruction can take up to five steps.

<table>
<thead>
<tr>
<th>Step</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>IF</td>
<td>Read an instruction from memory.</td>
</tr>
<tr>
<td>Instruction Decode</td>
<td>ID</td>
<td>Read source registers and generate control signals.</td>
</tr>
<tr>
<td>Execute</td>
<td>EX</td>
<td>Compute an R-type result or a branch outcome.</td>
</tr>
<tr>
<td>Memory</td>
<td>MEM</td>
<td>Read or write the data memory.</td>
</tr>
<tr>
<td>Writeback</td>
<td>WB</td>
<td>Store a result in the destination register.</td>
</tr>
</tbody>
</table>

- However, as we saw, not all instructions need all five steps.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Steps required</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>IF ID EX</td>
</tr>
<tr>
<td>R-type</td>
<td>IF ID EX WB</td>
</tr>
<tr>
<td>sw</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>lw</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

We will now study the implementation of a **pipelined version** of MIPS. We utilize the five stages of implementation for this purpose.
The PC is not shown here, but can easily be added.

Also, the buffer between the stages is not shown

The implementation of pipelining becomes "simpler" when you use separate instruction memory and data memory (We will explain it later). So we go back to our original Harvard architecture.
Pipelined MIPS

Why pipelining? While a typical instruction takes 3-4 cycles (i.e. 3-4 CPI), a pipelined processor targets 1 CPI (and gets close to it).

Pipelining in a laundromat -- Washer takes 30 minutes -- Dryer takes 40 minutes -- Folding takes 20 minutes. How does the laundromat example help with speeding up MIPS?

- Pipelining doesn’t help latency of single load, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup