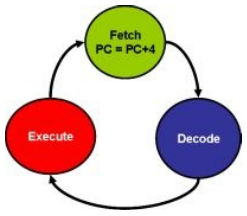
Single Cycle Data Path & Control



Purpose
Learn how to implement instructions for a CPU.
Method
Implement the datapath for a subset of the MIPS instruction set architecture described in the textbook using Logisim.

Files to Use
datapath.circ, control.circ, cpu32.circ, misc32.circ, and loop.mem.

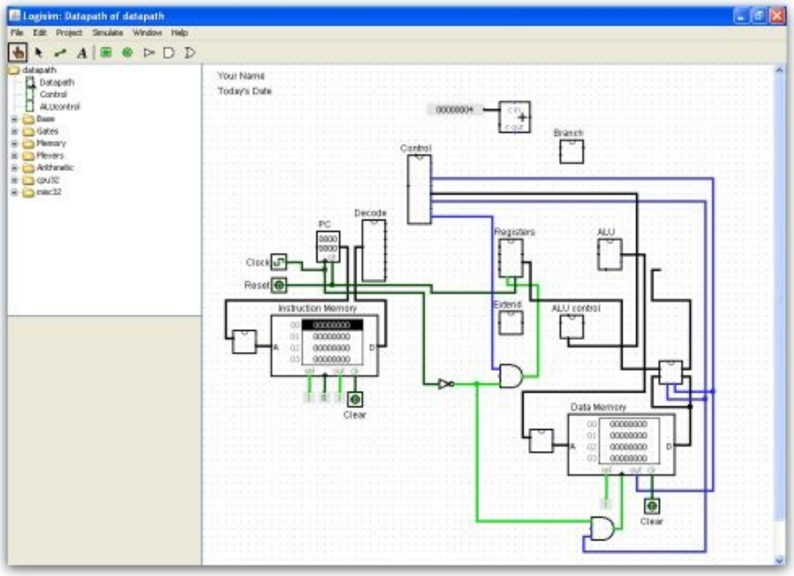
Acknowledgments: This assignment is an adopted version of an assignment constructed by Thomas M. Parks and Chris Nevison at Colgate University.

Get Started

Create a new folder (directory) named mips_datapath. Download all the neccesary files. Before you continue, make sure these files are in the mips_datapath folder:

- datapath.circ, control.circ, cpu32.circ, misc32.circ, loop.mem

Open the datapath.circ project in Logisim. You should see something similar to this:



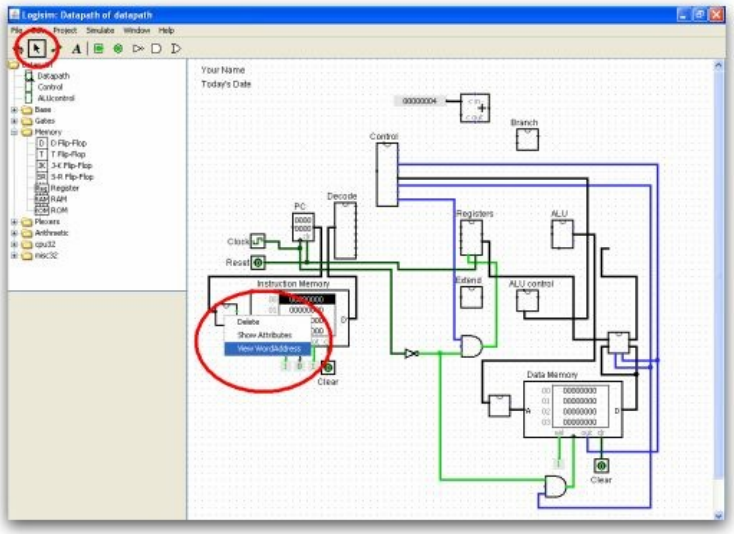
The `datapath.circ` project is very similar to figure 4.17, page 322 in the text book.

In this circuit there is hardware support for the following MIPS instructions: **add, sub, and, or, nor, slt, addi, lw, sw,** and **beq.**

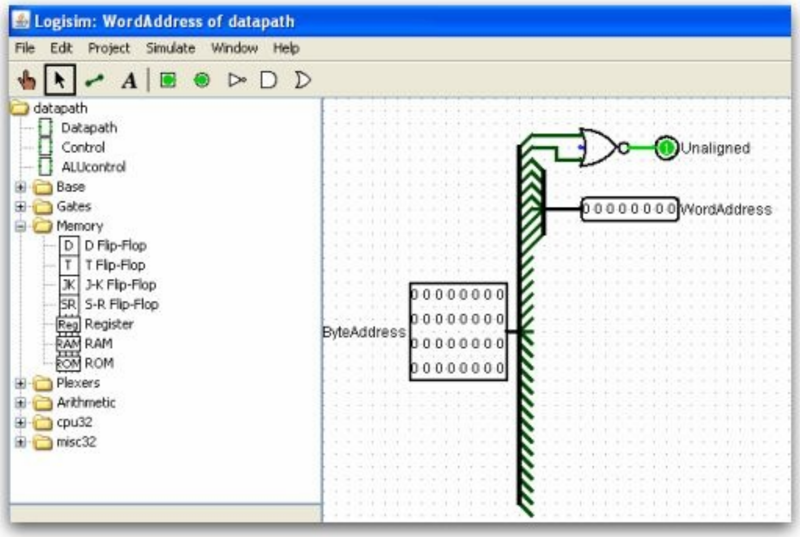
Logisim Components

Logisim comes with a few built-in component libraries. One such library is the memory library. From this library the RAM component is used for storing instructions and data. The register component is used for the Program Counter (PC).

In the datapath circuit you also see a number of square boxes. These boxes represents sub circuits. To the left of the instruction memory there is one such box. Right click on this box:



From the pop-up menu, choose "View WordAddress”:



This is a small sub circuit used to translate a 32 bit address to a 8 bit word address.

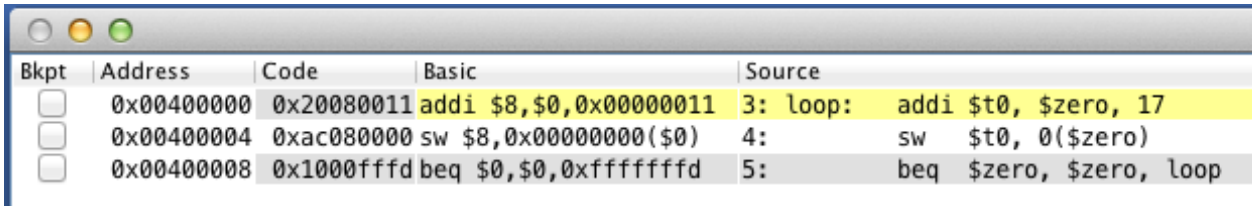
Step One: Create machine instructions

Right now the data path is not complete. Some wires are missing. Some more hardware needs to be added.

To start testing the data path, you should study this small MIPS assembly program:

```
loop:  addi   $t0, $zero, 17
       sw     $t0, 0($zero)
       beq    $zero, $zero, loop
```

To translate this to machine instructions you may use MARS:



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x20080011	addi $8,$0,0x00000011	3: loop: addi $t0, $zero, 17
☐	0x00400004	0xac080000	sw $8,0x00000000($0)	4: sw $t0, 0($zero)
☐	0x00400008	0x1000fffd	beq $0,$0,0xfffffffd	5: beq $zero, $zero, loop

Save the hexadecimal machine instructions to the loop.mem file.

NOTE: One instruction on each line. No 0x prefix.

Step Two: Understanding the control unit

To get a working datapath the control unit must send appropriate signals to various parts of the data path.

As preparation, study **figure 5.11** in the text book. In this figure you see a simple single cycle datapath for a subset of the MIPS architecture. Control signals such as ALUsrc etc are shown in blue writing. These control signals controls the behavior of the datapath.

In **figure 5.17** the main control unit is added. The control unit uses the operation field in the instruction to decide how to control the datapath by deciding which of the control signals should be enabled or not.

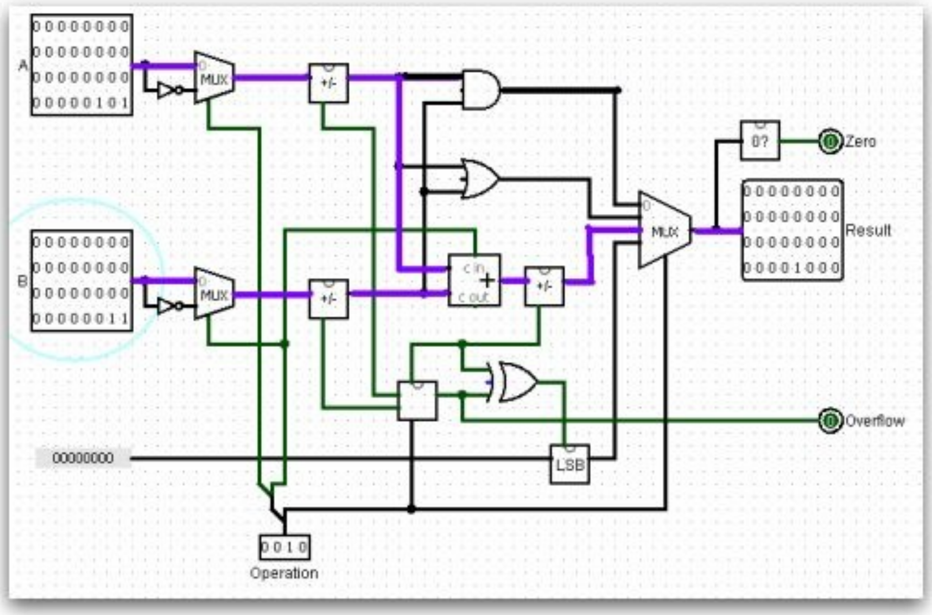
In **figure 5.18** you can see the settings needed for **R-type** instructions, **lw**, **sw** and **beq**. Look at the Logisim circuit and consult this table. Try to understand the values shown in the table - what choices in the datapath do these values represent.

NOTE: You don't need to construct a working control unit at this point. However, you should understand what signals the control unit must output to get a working datapath for the **addi**, **sw** and **beq** instructions.

NOTE: To control the datapath we will start off by manually emulating the control unit, that is, manually control the output from the control unit.

Step Three: The ALU

In **figure 5.15** ALU control is added. In the Logisim circuit, right click on the ALU and choose "View ALU":

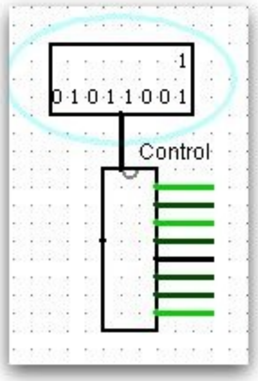


Make sure you understand how the Operation 4-bit input is used to control the function of the ALU.

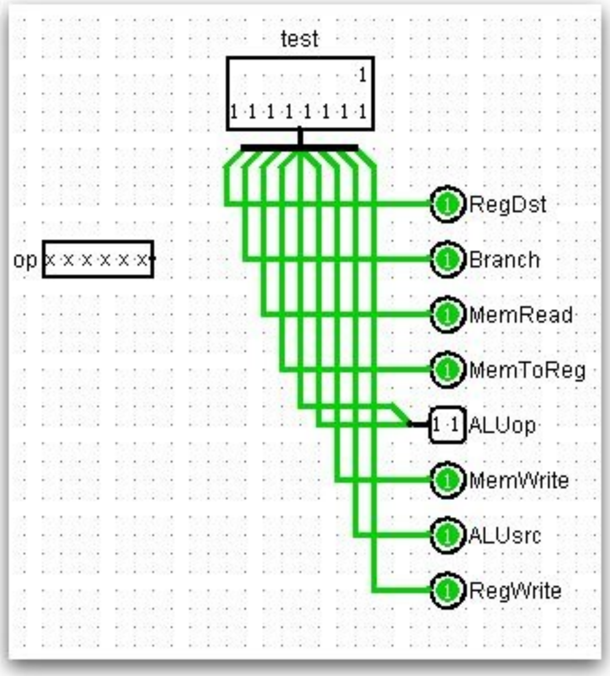
Step Four: Manual control

You don't have to construct a working circuit for control unit. But to get a solid understanding for how the control unit works, you shall manually control the output from the control unit.

To the top of the Control unit, connect a 9 bit input pin. Now you can use this input to manually control the control unit.



Open up the control unit circuit:



NOTE: All output lines from the control unit but the ALUop are 1 bits.

ALU: in a similar way as for the control unit, plug in a manual 4-bit input pin for manual control of the ALU control.

Fill in the following table. NOTE: some values don't need to be either 1 or 0, you may use X for don't care.

Table 1

	Reg Dst	**Branch**	**Mem Read**	**Mem to Reg**	**ALU Op (2 bits)**	**Mem Write**	**ALU Src**	**Reg Write**	**Operation (4 bits)**
addi									
sw									
beq									

Step Five: test of loop.mem

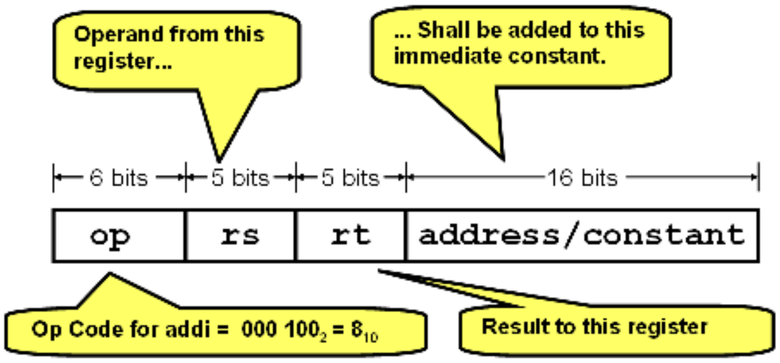
Right click on the instruction RAM memory. Load the machine instruction from the file loop.mem to the instruction memory.

Before you start to test your datapath using the small sequence of instructions loaded in to the instruction memory, you must be very careful:

- Make sure you loaded the machine instructions from loop.mem into the instruction memory.
- Make sure clock is high (green)
- Reset PC and the registers

addi $t0, $zero, 17

The first instruction in the small program (loop.mem) is an I-type instruction:

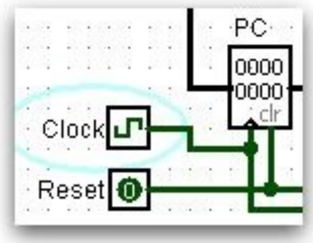


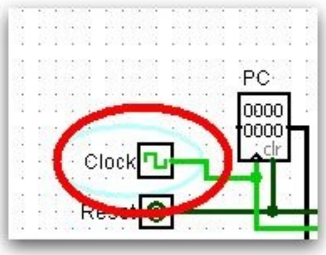
Operand from this register...
... Shall be added to this immediate constant.
6 bits
5 bits
5 bits
16 bits
op
rs
rt
address/constant
Op Code for addi = 000 100$_2$ = 8$_{10}$
Result to this register

Manually, set the output values from the control unit (use the table).

- Add wires and possible some hardware (gates, multiplexers etc) to make the data path execute the addi instruction.

For data to be stored to a register or to memory, the clock signal to *these units* must change from low to high (rising edge trigged). Remember that you started with the clock high:



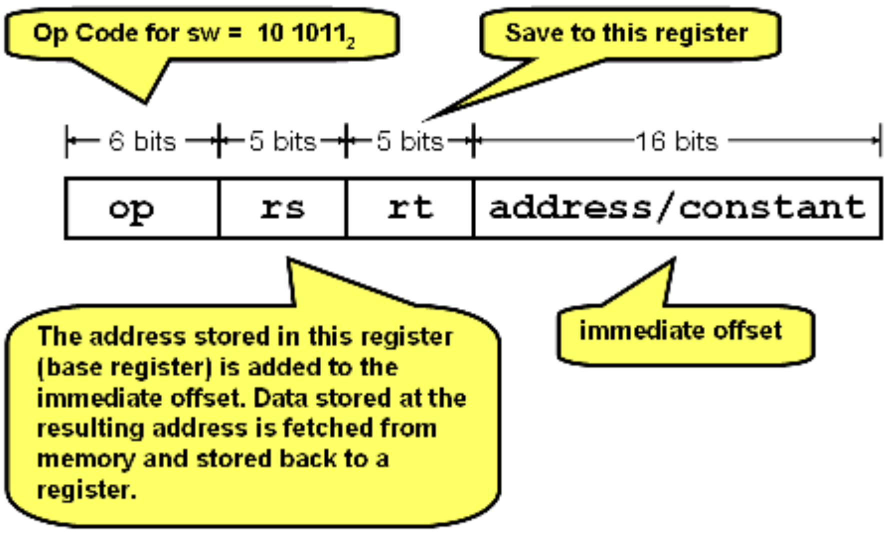


Click on the clock to change from high (green) to low (black)

NOTE: In the circuit you can see that the clock signal sent to the registers (and to the data memory) is inverted. Why do you think this inversion is needed?

sw $t0, 0($zero)

The store word instruction is an I-type instruction:

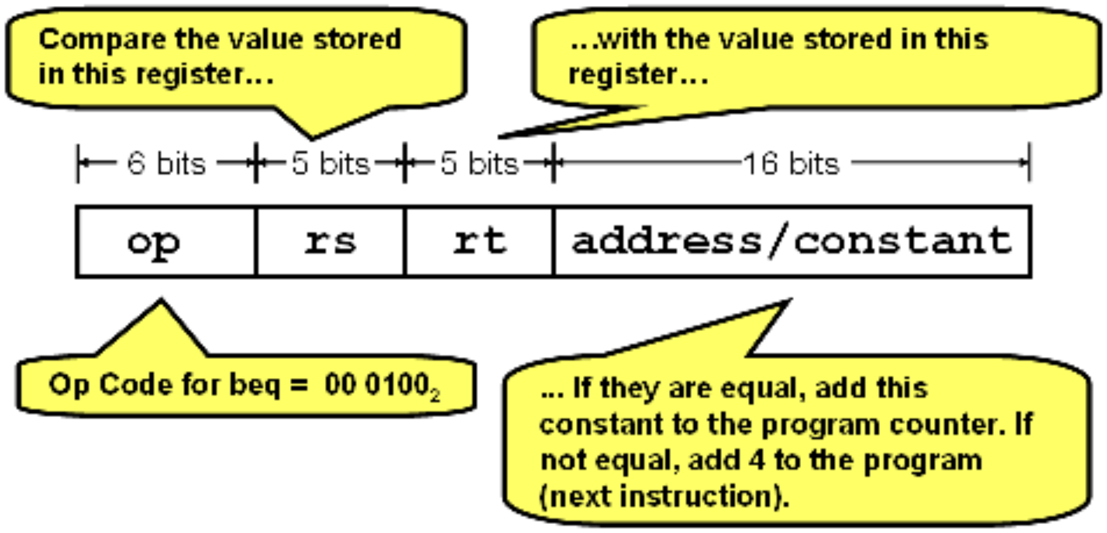

Op Code for sw = 10 1011_2
Save to this register
6 bits
5 bits
5 bits
16 bits
op
rs
rt
address/constant
The address stored in this register (base register) is added to the immediate offset. Data stored at the resulting address is fetched from memory and stored back to a register.
immediate offset

For this instruction the ALU is used to calculate the final memory address.

- Add wires and possible some hardware (gates, multiplexers etc) to make the data path execute the sw instruction.

beq $zero, $zero, loop

The last instruction in the small program, beq, is an J-type instruction:

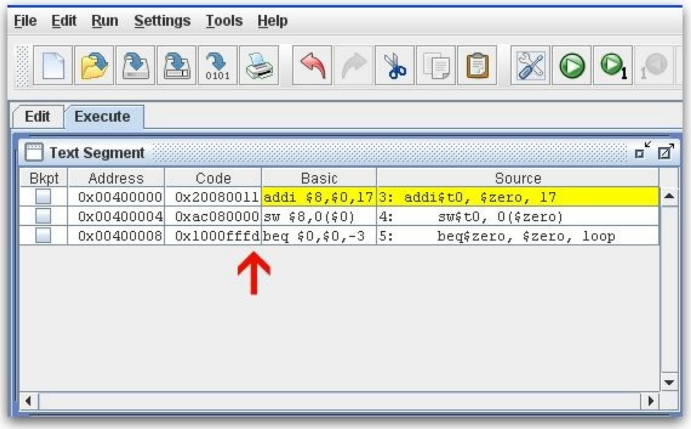

Compare the value stored in this register...
...with the value stored in this register...
6 bits
5 bits
5 bits
16 bits
op
rs
rt
address/constant
Op Code for beq = 00 0100_2
... If they are equal, add this constant to the program counter. If not equal, add 4 to the program (next instruction).

To compare the two registers the ALU can be used. For two number A and B, A-B == 0 IFF A == B.

NOTE: there is a zero bit output from the ALU that will be set to 1 when the result from the ALU is 0. This signal can be used to decide whether or not we shall jump (PC = PC + Offset) or continue executing the next instruction (PC = PC + 4).

The machine instructions produced - the assembler produces a branch offset that is encoded in the branch instructions as an immediate 16-bit constant. This constant is set to the number of instructions to jump if the branch condition is true.

If we use MARS to produce the machine instructions we get the following result:



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x20080011	addi $8,$0,17	3: addi$t0, $zero, 17
☐	0x00400004	0xac080000	sw $8,0($0)	4: sw$t0, 0($zero)
☐	0x00400008	0x1000fffd	beq $0,$0,-3	5: beq$zero, $zero, loop

- Add wires and possible some hardware (gates, multiplexers etc) to make the data path execute the beq instruction.

Make sure your datapath can loop correctly.

Non mandatory part - adding more functionality

So far the datapath only implements a subset of all MIPS instructions. To support more instructions more hardware must be added to the datapath.

- Extend the circuit and add support for the sll (shift logical left) instruction.
- Make sure you can exectute the jal (jump and link) instruction correctly (updates $ra).
- Make sure you can return from a subroutine wiht jr $ra

Demonstrate by manually controlling the control unit how to execute a jump and link followed by a jump register $ra