CPS311 Lecture: CPU Implementation: The Registers, ALU and Data Paths

Last revised October 23, 2019

Objectives:

1. To show how a CPU is constructed out of a clock, register set/ALU/datapaths and a control unit.
2. To discuss typical components of the register set/ALU/datapaths
3. To show how a mips-like machine could actually be implemented using digital logic components already seen

Materials:

1. Projectable of overall block diagram of CPU
2. Projectable of Block Diagram - Single-Cycle Implementation modified to exclude details of memory, operation control, and RTL
3. Projectables of
   a. A typical bit of a register
   b. Implementation of an output of the register set
   c. Implementation of decoder for enables for register set
   d. Implementation of a typical bit of the ALU
4. Java Single Cycle MIPS implementation simulation
5. Programs for the above
   a. Double, then add 1, to memory location 1000
   b. Delayed branch demo
6. Circuit Sandbox simulations
   a. One bit of a register
   b. Four bit register
   c. Register set
   d. ALU Bit
   e. 4-bit ALU
   f. Four bit register with 2 inputs
7. Java Multicycle MIPS implementation simulation
8. Programs for the above
   a. Add 1 to memory location 1000
   b. Machine language code for Lab 5 Part I
9. Handout: RTL for multicycle MIPS implementation
10. Handout with assembly language and machine language versions for multicycle programs

I. **Introduction**

A. For the last several weeks, we have been focussing on computer architecture. Today (and in fact for the rest of the course) we turn out attention to computer organization. What is the difference in meaning between these two terms?

ASK

1. Computer architecture refers to the functional characteristics of the computer system at the ISA level, as seen by the assembly/machine language programmer (or the compiler that generates assembly/machine language code), as the case may be.

2. Computer organization refers to the physical implementation of an ISA.

3. Historically, significant architectures have had numerous implementations, often over a period of decades.

   a) IBM mainframe architecture - first developed with System 360 in mid 1960's - still being used (with modifications) in machines manufactured today.

   b) DEC PDP-8 architecture - first developed in late 1960's - last implementation in 1990. (Went from minicomputer with CPU realized as discrete chips to microprocessor).

   c) IBM/Motorola PowerPC architecture (the chip once used in Macintoshes and once (still?) used in CISCO routers, many video game consoles and TV set-top boxes) - first developed in mid 1990's, still utilized today. (The most recent version, the G5, represents the 5th generation of this architecture. The G5 is a 64-bit chip that fully implements the 32 bit architecture with 64-bit extensions Each generation has had multiple implementations.)

   d) Intel IA32 architecture - first used in 80386 family in mid-1980's; the 64-bit chips used in virtually all PCs are still backwards compatible with this architecture. (Often known as x86)

B. Of course, a complete computer system consists of a CPU, Memory, and IO facilities - possibly all on the same chip in embedded systems, or on multiple

chips. For a while, we will focus on the CPU - we will address memory and IO later. In early computers. the CPU was built up out of multiple discrete components, but today the CPU is a single chip. However, we will look at the internals of the CPU in terms of digital devices we have discussed earlier such as gates, flip flops, multiplexers etc - realizing that today these are all realized on a single chip.

C. To try to develop in any detail the implementation of a contemporary CPU is way beyond the scope of this course - and also way beyond the scope of my knowledge, in part because manufacturers don't publish all the details about their implementations - for obvious reasons! Instead, we will focus on some hypothetical implementations of a subset of the MIPS ISA - which is relatively simple, and for which published information actually is available.

1. It should be understood from the outset that the implementations presented here are definitely **NOT** the structure of an actual MIPS implementation.

2. For pedagogical reasons, the two implementations presented in this lecture are quite different from the way MIPS is actually implemented. (One we will present later in the course is much closer to the actual implementation, but is still much simpler.)

3. The implementations we will present does not support a number of features of the MIPS ISA - though these could be added at the cost of additional complexity.

   (a) The hi and lo registers, and multiply and divide instructions.

   (b) Support for coprocessors, including floating point instructions.

   (c) Kernel-level functionality, including interrupt/exception handling.

   (d) The distinction between signed and unsigned arithmetic - we will do all arithmetic as signed.

   (e) Byte and halfword operations.

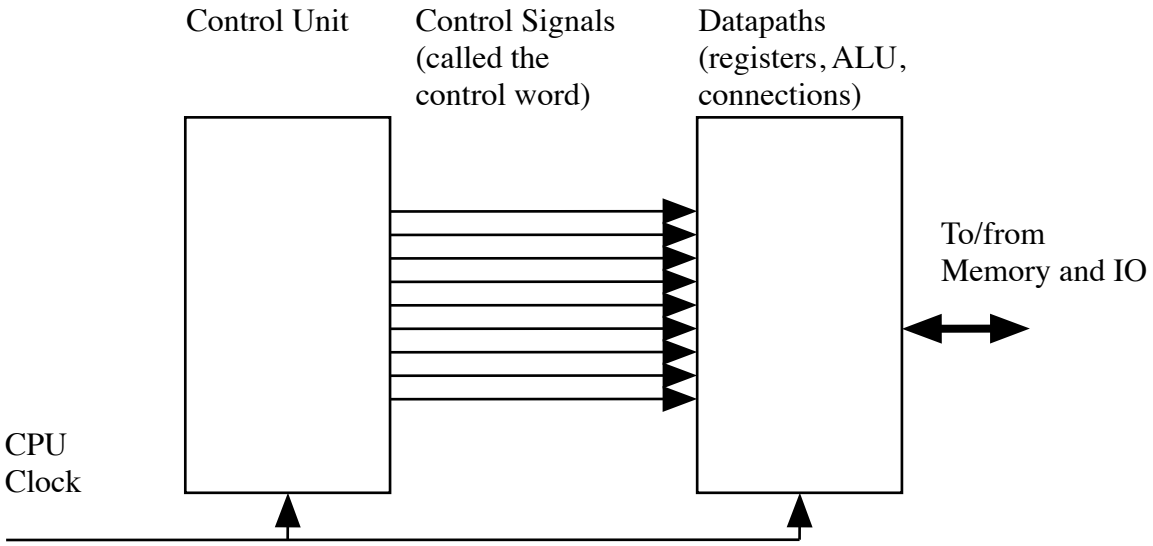
4. The implementations we will present do not include some efficiency "tricks".

D. To understand CPU implementations, we make use of a fundamental principle in computer science: the notion of levels of abstractions.

1. In essence, what this means is that we can look at any given system at several different levels. Each level provides a family of primitive operations at that level, which are typically implemented by a set of primitive operations at the next level down.

2. Today, we will be doing this at the hardware level. Earlier in the course we saw, for example, how a JK flip flop could be viewed as an abstraction having a certain transition table etc. - but that abstraction an be realized, in turn, by a network of gates, each of which can be realized by patterns deposited on a silicon chip (or relays, or tinker toys!)

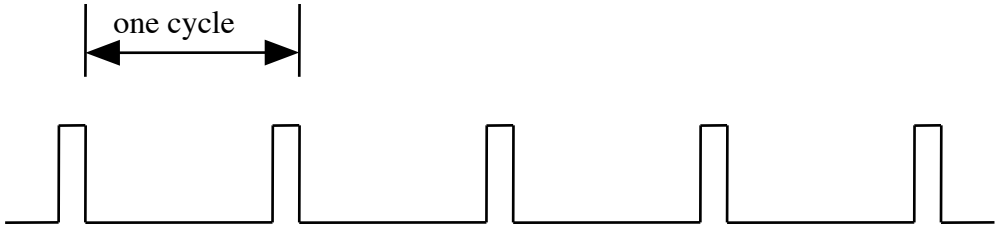
E. A CPU can be regarded as having the following overall structure:



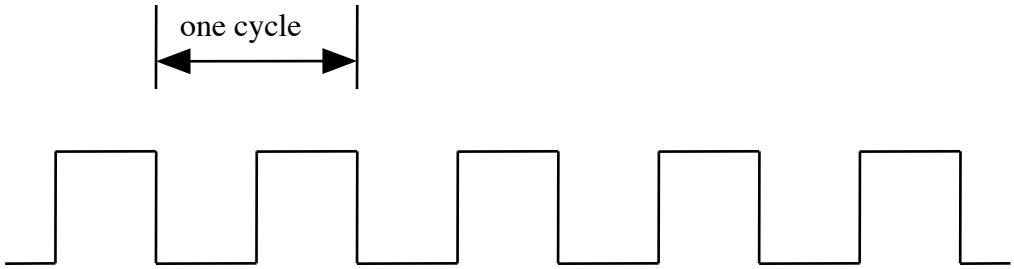
PROJECT

1. The portion on the right contains the visible registers that an assembly/ machine language programmer sees - e.g. the PC and 32 general registers in MIPS. It also contains an arithmetic-logic unit and data paths that perform required operations on the registers - e.g. adding two registers. This portion will be our focus in this lecture and the next.

2. The clock generates a regular series of pulses that synchronize state changes in the registers. Its output looks like this:



or perhaps this:



a) The frequency of the clock dictates the overall speed of the system.

(1)For example, if a computer is reported to use a CPU with a 2 GHz clock, it means that there are 2 billion clock cycles per second - so each cycle takes 1/2 nanosecond.

(2)The maximum clock frequency possible for a given system is dictated by the propagation delays of the gates comprising it. It must be possible for a signal to propagate down the most time-consuming path in not more than one clock cycle.

(3)Most systems are engineered conservatively, in the sense that the clock frequency is actually slightly slower than what might actually be possible. This allows for variations in component manufacture, etc. It also leads to the possibility of overclocking a given CPU as a (somewhat risky) performance-improvement “trick”.

b) The various registers comprising the system are synchronized to the clock in such a way that all state changes take place simultaneously, on one of the edges of the clock.

In the examples we will be developing, we will assume that all state changes take place on the falling edge of the clock. (It would also be possible to design a system in which state changes take place on the rising edge of the clock.)

In some systems (including most mips implementations), while most state transitions take place on one edge, there are some transitions that occur on the other edge. This allows certain operations to be allocated 1/2 a cycle of time. (But more on this later - for now we ignore this possibility.)

3. The control unit generates control signals that control the operations taking place in the datapaths.

a) This includes things like signals that control what computation the ALU does (add, subtract, and, or ...); load enables to the registers that determine whether a register will change state on the next clock pulse, etc.

b) The set of control signals, together, is sometimes called the control word.

c) A new control word is generated prior to each clock pulse, specifying what operations are to be performed on that clock pulse.

d) We will consider the implementation of the control unit portion of the CPU in a subsequence series of lectures.

II. The Registers, ALU, and Datapaths

A. This portion of the CPU includes the circuitry for performing arithmetic, and logic operations, plus the user visible register set and special registers that connect to the Memory and IO systems. The actual structure of this part of the CPU as physically implemented is usually not identical to that implied by the ISA.

1. The actual physical structure that is implemented is called the microarchitecture.

2. The microarchitecture must, of course, include components that correspond to the various parts of the system that appear in the ISA (e.,g. the registers). We call this the architectural state.

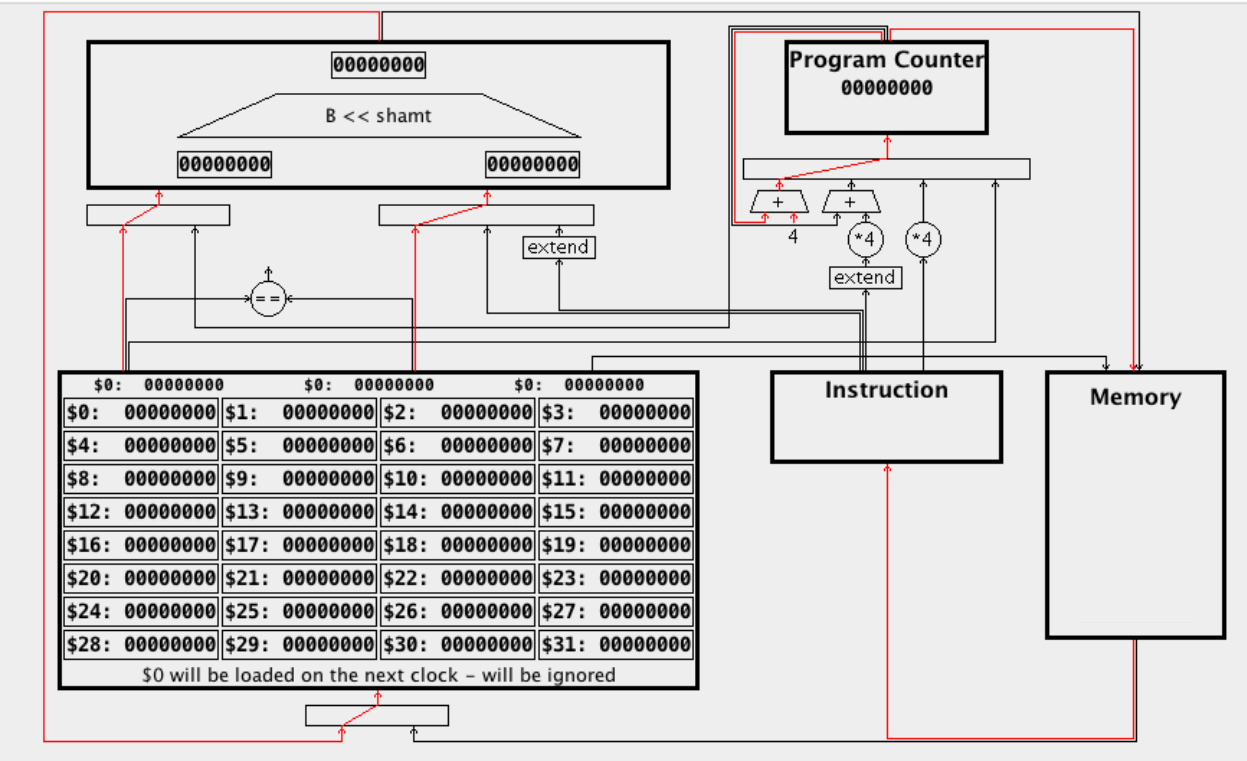
3. The microarchitecture usually includes registers that do not appear in the ISA. We call these the non-architectural state.

4. An ISA might have various specialized registers, but the microarchitecture might utilize general registers which are mapped to the various special functions in the ISA.

5. It is common today to find CPU's that have a CISC ISA being implemented by a RISC microarchitecture (RISC core) We will not, however, pursue this topic since things can get quite complex!

B. The following is a block diagram of the Datapaths for the single-cycle MIPS simulated implementation we will be discussing today.

PROJECT Block Diagram for MIPs Single Cycle Implementation



1. In this simulation, there is a one-to-one correspondence between most of the components of the architectural state and components in the microarchitecture.

   a) The Register Set (lower left corner) holds the 32 general registers visible to the assembly/machine language programmer or the compiler.

   b) The ALU (upper left corner) performs various primitive operations on 32-bit values - e.g. add, subtract, and, or ...

   c) The Program Counter (PC) is a 32-bit register that holds the address of the next instruction.

2. The Instruction Register (IR) holds the instruction currently being executed. It is part of the non-architectural state, since it is not directly visible in the ISA (and another implementation may handle it differently.) It is utilized by the control unit to determine what operations need to be

performed, but also provides some information to the ALU, such as constants used for I and J format instructions.

a) At the beginning of a cycle, the IR will hold the instruction being executed on that cycle.

b) At the end of a cycle - when the register updates called for by that instruction are done - the IR itself will be updated to hold the next instruction.

3. The Memory is shown because data flows to/from it from the portion we are focussing on. It has to have two ports - one for data and one for instructions - since on each cycle it fetches an instruction and in many cases it may also need to read or write data - hence the two addresses coming in from the PC and ALU, two data lines going out to the IR (for instructions) and the register set (for reads), and one data line coming in from the register set (for writes).

4. The lines connecting the various components are data paths along which data can flow from one component to another.

   a) In most cases, they are 32 bits wide, drawn as a single line.

   b) They are always one-way (note the arrow heads).

   c) In the simulation, they are shown in red if the are currently active, and black if they are not.

5. The small rectangles below the register set, ALU, and PC are MUXes that allow one of several inputs to be selected (as specified by the control unit.) (The line in the MUX indicates which source of input is currently selected.).

   The small "+" and "*" boxes feeding into the MUXes are adders or multipliers (left shift two places). The boxes labeled "extend" are sign extenders - converting a 16 bit value into 32 bits by replicating the sign of the 16 bit value into the bits of the upper half.

a) The register set can receive data either from the ALU or memory - the MUX determines which. It receives data from the ALU when performing an R-Type operation such as add, and from memory when performing a load operation.

b) The ALU receives two inputs.

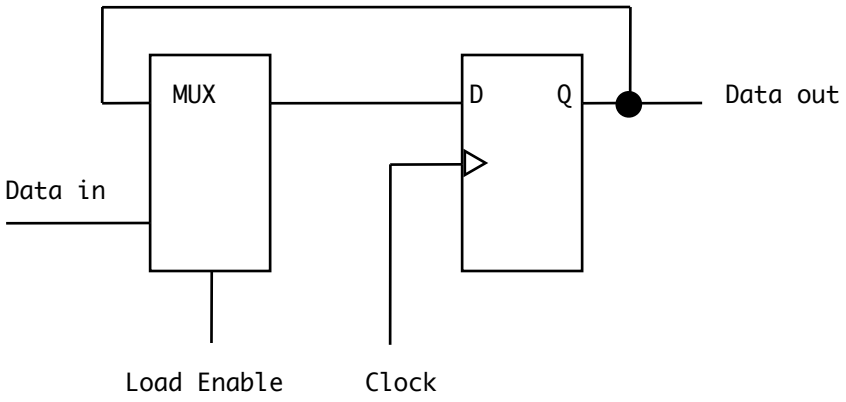
(1)One can come either from the register set or from the PC. The former is used for most operations, with the latter being used just when executing the JAL instruction (which copies the PC into a general register).

(2)The other can come either from the register set, or from the constant field of the current instruction. In the latter case, it may or may be sign-extended (extended for operations like add; not extended for operations like and).

c) The PC holds the address of the next instruction, and is updated as this instruction is fetched. It can receive either 4 added to its current value, (most instructions) or 4 * the I constant of the current instruction added to its current value (branch instructions) or the J field of the current instruction (Jump and JAL).

C. Implementation of Individual Registers.

1. A 32-bit register - such as the ones in the Register Set, the PC and the IR - can be realized using 32 combinations of a flip flop and a 2-way MUXs - one for each bit.



PROJECT

DEMO Circuit Sandbox - Single Register bit (Clear using bottom button to start run)

2. Data in connects to the data path going into the register, and data out to the data path going out of the register.

3. All the register flip flops (over 1000 of them) are connected to the clock.

4. Thus, all the flip flops load a value on each clock. However, the MUX at the input can arrange for this to either be a copy of the current value (hence no change) or an input coming in. The load enables for all the flip flops in a given register are connected together - when it is 0, the register retains it value; when 1, the register loads a new value from the 32-bit input data path.

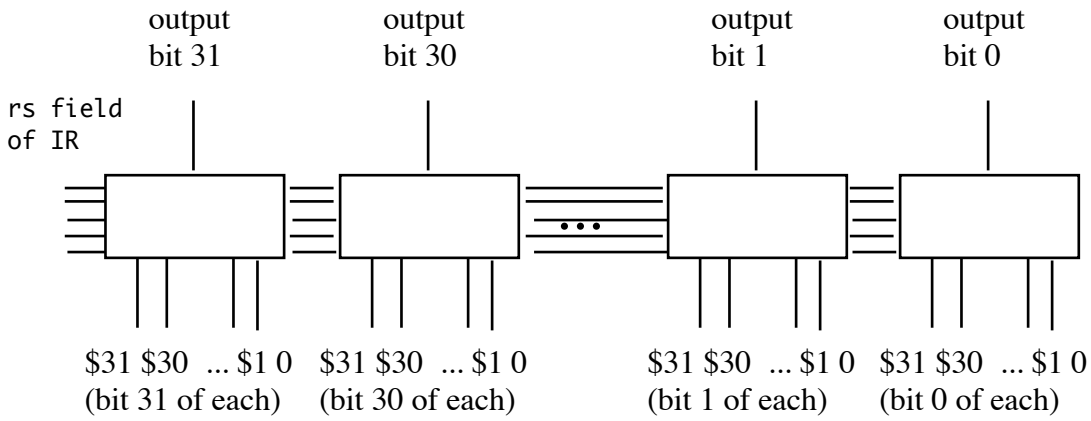
   DEMO: Circuit Sandbox - 4 bit register. (Run program from copy in Circuit Sandbox Demos folder to get special component. Operation is controlled by three switches on top: 000 = and, 001 = or, 010 = xor, 011 = nor, 111 = add)

D. Implementation of the register set.

1. The register set contains 32 registers, each composed of 32 flip-flop/MUX pairs - (except for $0, where all of the bits can simply be 0)

2. The register set furnishes three outputs - two to the ALU and one to memory - controlled, respectively, by the rs, rt, and rt fields of the current instruction. Each output can be realized by 32 MUXes, each with 32 data inputs and 5 selection inputs.

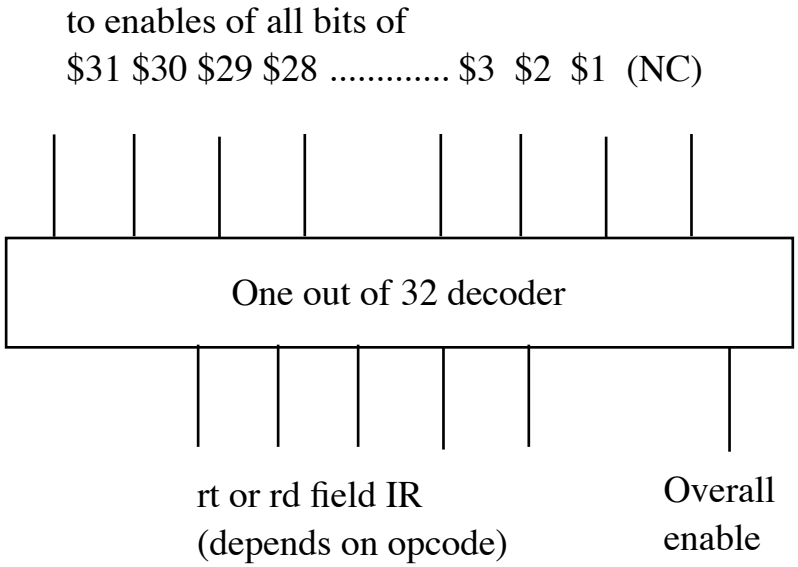
For example, the left most (rs) output may look like this



PROJECT

3. The corresponding data inputs of each register may be connected to the data input to the register set - e.g. bit 31 of each of the registers ($1..$31) may all be connected to data input bit 31, etc.

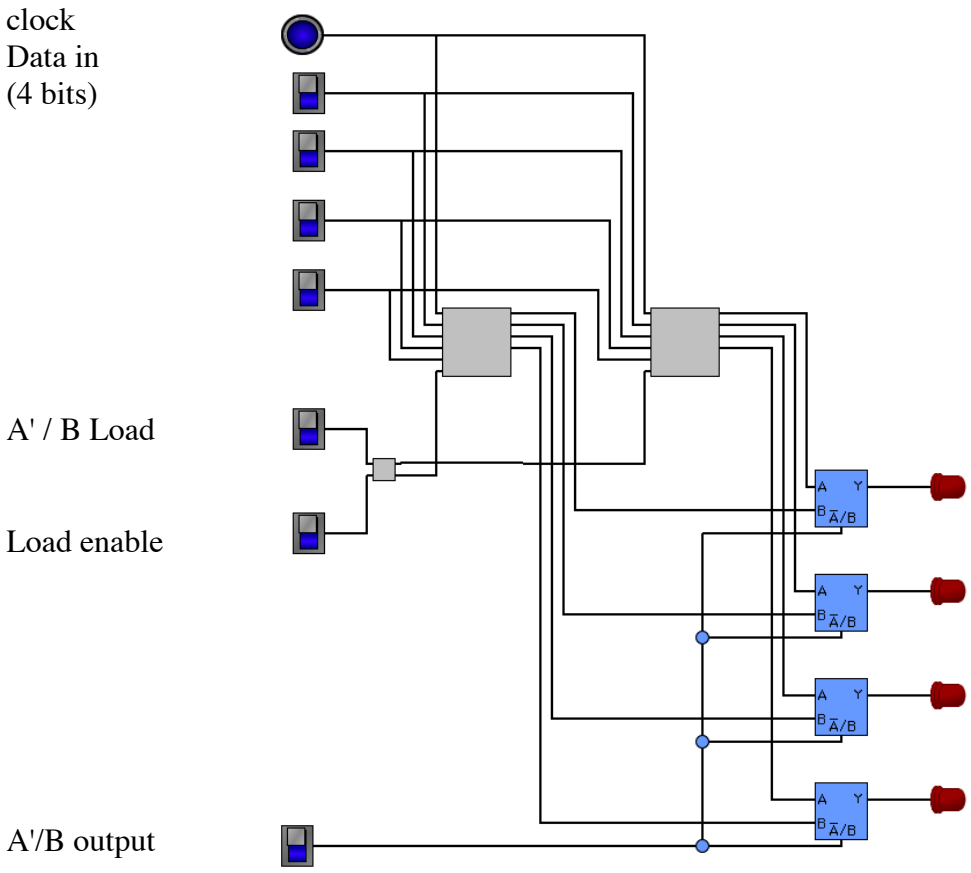
4. The enables of each register may be connected to a 1 out of 32 decoder that selects which bit gets loaded based on a 5 bit value (either the rt or rd field of the current instruction.) - e.g.



PROJECT

5. Thus, the operation of the register set is controlled by the rs, rt, and rd fields of the current instruction plus a single load-enable bit of the control word!

   CIRCUIT SANDBOX MODEL register set (2 4-bit registers)



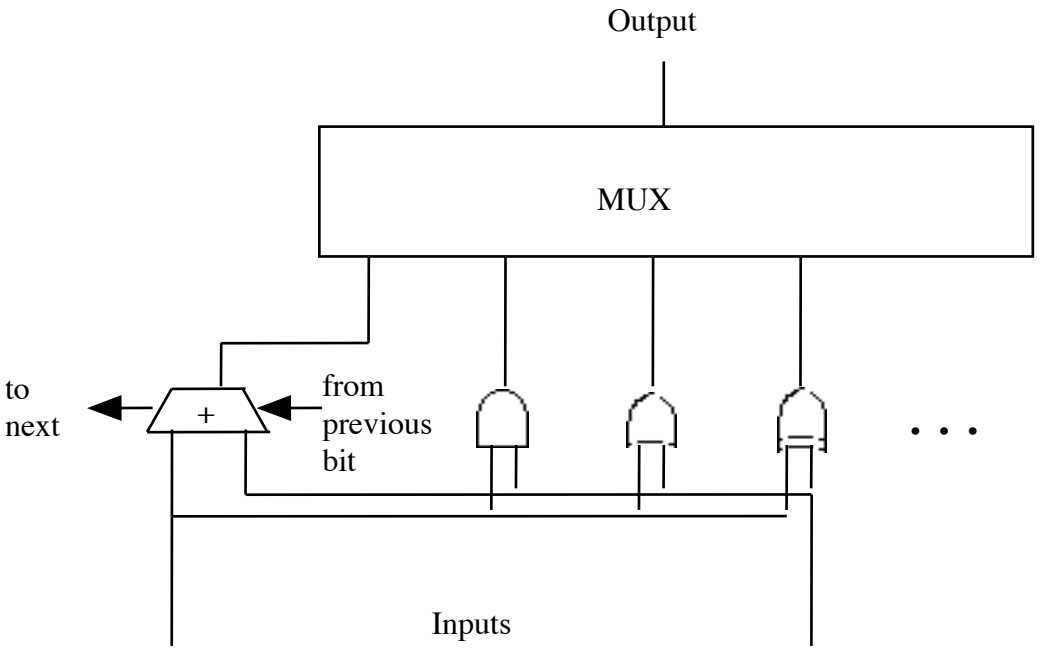
E. The ALU of MIPS performs one of 10 different operations on two 32 bit input values to produce a 32 bit result - with some additional variations for the shifts. It might implemented by 32 copies of a circuit consisting of a MUX plus appropriate gates/gate networks for each function.

1. The ALU needs to be capable of performing the following operations

   Output ← A + B
   Output ← 1 if A < B (slt)
   Output ← A & B
   Output ← A | B
   Output ← A ^ B
   Output ← B << 16
   Output ← A (B ignored)
   Output controlled by funct field of IR

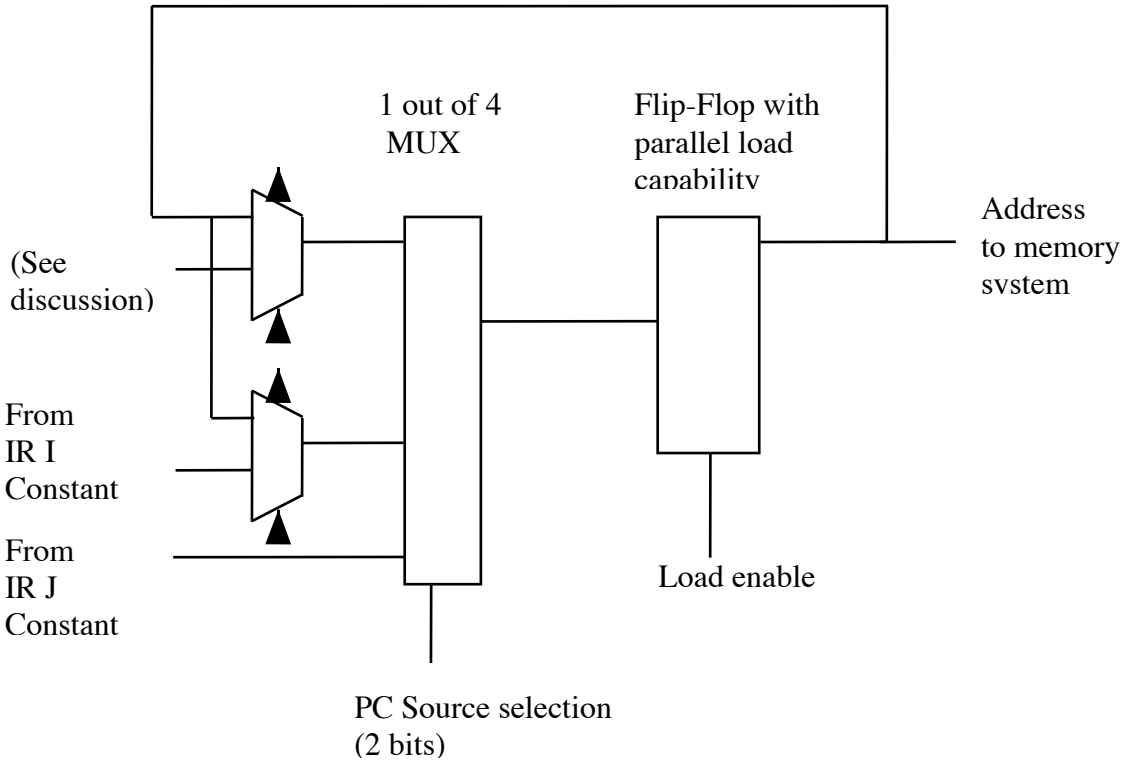
2. Thus, a typical bit (replicated 32 times) might look like this:

   PROJECT



   DEMO: Circuit Sandbox simultations of ALU Bit and 4-bit ALU

F. A typical bit of the PC could be implemented like this:



PROJECT

DEMO Circuit Sandbox Simulation of a register with 2 inputs

1. Adding + 4 is achieved by hard-wiring the second input of bit 2 of the adder to be 1, and all others to be 0.

2. Scaling of constants from the IR is done by shifting - e.g. bit 0 from the IR goes to the MUX/flip flop for bit 2, bit 1 from the IR goes to bit 3, etc. Bits 0 and 1 always receive 0. For j/jal, bits 31..28 receive the bit in the corresponding position in the PC, since the constant is 26 bits shifted left two places to produce a 28 bit constant.

3. The adders for PC+4 and adding the I constant receive carry from the previous bit and pass carry to the next bit, indicated by the arrows going into and out of the adders.

4. Actually, since the PC must always contain a multiple of 4, it is not necessary to implement the two low order bits as flip-flops; they can simply be hardwired to 0.

5. In this implementation, the PC is loaded on every cycle, so no load enable is needed (or it is hardwired to 1).

G. Implementation of the IR is straightforward - it is just a simple register, with the inputs connected to the memory, the outputs controlling various functions - and going to the control unit - and the load enable always 1, since the first step in executing an instruction is fetching it! (Which, in this simulation, is actually done at the end of executing the previous instruction.)

H. We have now seen that every component of the datapaths portion of the CPU could be built up from components we are already familiar with - flip flops, multiplexers, decoders, and simple gates.

1. In the earliest days, each of these components had to be built up out of basic electronic components such as vacuum tubes or transistors. CPU's implemented this way might fill a room!

2. Later, CPU's were built up out of individual chips like the ones you've used in lab - each containing one or two flip flops, or multiplexes, etc. At this point, a CPU might fill one or more circuit boards.

3. Today, CPU's are implemented on a single chip - which means that each of the component parts is implemented in an area on the chip. But even though you cannot see the individual pieces, they're still there.

I. As we have noted, the components are connected by data lines and MUXes.

III.Controlling the Operation of the Data Paths

A. In our discussion of how the various parts of the data portion of the CPU are built, we've noted that the specific operations performed are controlled by various bits of the control word. Let's pull these all together. It turns out that we need just 14 control bits to control everything - all based on the content of the IR!

1. 2 bits to control the updating of the PC (PC + 4, branch, or jump) - one possibility unused.

2. 2 bits to control reading from or writing data to memory (these actually go to the memory system, not what we have just been discussing.)

3. 1 bit to control the left source to the ALU (register specified by rs or PC)

4. 2 bits to control the right source to the ALU (register specified by rt or immediate constant or sign-extended immediate constant) - one possibility unused.

5. 3 bits to determine the operation performed in the ALU along with the funct field in the instruction. (For the immediate instructions, the funct field in the instruction is actually part of the constant.)

6. 1 bit to control whether a new value is loaded into a register in the register set.

7. 1 bit to control where this new value comes from (if one is being loaded) - the output of the ALU or memory.

8. 2 bits to control which register is loaded (determined by rd field of instruction, determined by the rt field, or register 31 (required for JAL))

B. Recall the notion of hierarchies of abstraction. At this point, we can describe what is happening using a level of abstraction called the Register-Transfer Level. This level has the following characteristics:

1. The primitive operations at this level are called <u>microoperations</u>

2. Each microoperation is directly realized by hardware, and can be carried out in a single clock cycle.

3. Each machine-language level instruction in the ISA is realized by a series of microoperations.

   a) Some of these may be done in parallel if they use disjoint hardware components

b) But most will need to be done sequentially, resulting in several clock cycles being used to implement the operation from the ISA.

4. An RTL operation is described by an expression of the form

   `destination ← source`

   which indicates that, on the clock pulse, the source value is copied into the destination.

   a) The source may be

      (1)A register

      (2)Some simple combinatorial operation performed on two registers - e.g. bitwise &, |, ^; add or subtract. (But not multiply or divide)

      (3)A memory cell. (Often abbreviated as M[x], where x is the source of the address)

      NOTE: Actual access to a cell in memory may take 100's of clock cycles. However, as we will see later in the course, the memory system is configured to allow most accesses to be done in one clock cycle. Thus, a microoperation involving a memory cell may be done in one clock, or may result in the CPU being stalled until it can be completed.

   b) The destination may be

      (1)A register

      (2)A memory cell (same notation and caveats as above).

5. Sometimes a microoperation is done on just part of a register. In this case, the specific bits are indicated in parentheses after the register name - e.g.

```
someReg(0) ← 0
someReg(7..0) ← someOtherReg(15..8)
```

6. Associated with each RTL operation is a control signal that determines whether or not that operation is performed on a particular clock. This may be denoted by

   ```
   signal name : operation
   ```

   which indicates that the operation is performed just when the control signal in question is true

7. If two or more microoperations involve disjoint sets of hardware components, they can be done in parallel on the same clock. This is quite common in CPU hardware. This is denoted by writing the two microoperations on the same line, separated by a comma - e.g.

   IR ← M[PC], PC ← PC + 4

C. We can now see how the data portion of the CPU we have just described can actually implement machine-language operations in the ISA.

1. First demo: Consider a a single instruction, which adds the contents of registers 5 and 6 and puts the result in register 4:

   ```
   add $4, $5, $6
   ```

   ```
   000000 00101 00110 00100 00000 100000
   0x00a62020
   ```

   DEMO: Put the instruction in memory location 0, be sure the PC is 0, set $4 in 0, set $5 to 1, set $6 to 2
   Discuss RTL and data paths
   Set Visible Delays on, Clock and watch run, then look at $4

2. Now let's consider a simple program: Add 1 to the value stored in memory cell 1000 and then write it back.

   Obviously, this is a far from interesting (or useful) program - but it will allow us to see how a series of machine instructions are executed by hardware.

a) This could be realized by the following program. (It ends with a nop to mark the end of the program after the last instruction has been executed.)

```
lw $2, 1000($0)
addi $2, 1
sw $2, 1000($0)
```

b) In machine language, this is realized by the following code:

PROJECT Add 1 to memory location 1000

c) DEMO Execution, walking through RTL and enabled data paths at each step - then show final result.

D. This implementation does have one idiosyncrasy. Consider the execution of a conditional branch or jump instruction.

1. The PC always contains the address of the next instruction - which during the execution of a branch/jump will be the address of the next word in memory, not the target of the branch or jump, since the PC is updated at the same time the IR is.

2. At the next clock tick, the PC will be updated to the target address calculated by the branch/jump instruction, but since the IR and PC are both updated in parallel, the IR will load the word just after the instruction - not the branch target. As a result the instruction just after the branch/jump will always be executed before we being executing at the target address.

3. DEMO: Program consisting of a single "branch to self" infinite loop followed by an addi $1, 1.

a) Show Program.

b) Step through the final loop several times and note how $2 changes each time through. (When run with the Single Cycle implementation, needs to be clocked once initially to load the IR the first time)

c) Let it run.

4. This phenomenon is called delayed branch. Since the single cycle simulation we have done thus far is not the way MIPs is actually implemented, this is a non-issue at this time. But it will reappear when we get to the pipelined implementation that actually is used! (And it can be dealt with fairly easily - we'll discuss later.)

IV.Multi-Cycle Implementation

A. Though the implementation just developed works reasonably well for MIPs, there are serious practical issues that arise in many cases.

1. Timing issues

a) Recall that physical devices need some amount of time for performing an operation - therefore, there is a finite delay between the time the input to a device changes and the time the output of the device is correct.

b) Since the input of an operation is often the output of the previous operation, clock rate is limited by the time needed for all the operations in a sequence to be executed. Since the clock rate is not dependent on what instruction is being executed, the limit turns out to be the longest time for any instruction.

MIPS Example: An lw instruction performs the following operations, each of which depends on the result of the one just before.

Add register specified by IR and constant in IR in ALU
Fetch data from memory from address specified by ALU output
Store value fetched into a register

DEMO: Simulation with delay injected. (Start from pc = 0 and IR = 0 and just step - first clock will load first real instruction and do nothing else since nop)

c) While this is not a major issue with MIPs since all instructions encounter similar logic delays, it can be a significant issue for architectures where instruction times vary widely - e.g. on a machine having multiply as a regular instruction a multiply takes much longer than an add.

2. Multiple use of functional units issues

a) On MIPs, the data memory and ALU are used just once per instruction.

b) But some machines may use these units more than once on some instructions.

Example; A one-address machine using an address mode like displacement mode on an add instruction may use the ALU once to calculate the address and again to perform the addition.

Example: A one-address machine doing an addition on an operands memory may need to read, then modify, then write a location in memory.

Example: A two-address machine doing an addition may need to read two operands, perform a computation, and then write the result.

3. Instructions that contain loops.

Example: The x86 architecture includes several instructions that operate on character strings, such as copying or comparing them. These instructions perform a single computation for each character in the string(s) involved - with the number of computations needed being dependent on the length of the string.

B. To address issues like this, it is possible to break an instruction into smaller steps, and execute the instruction as a series of individual steps. This uses multiple clock cycles - but if the steps are made small enough, the overall time (number of steps * clock length) can be comparable to single cycle time for short instructions, and can easily vary from instruction to instruction so that some instructions complete in fewer cycles than others.

C. Consider a MIPs implementation using this idea. Again, this is not the way MIPs is actually implemented, but it will help us understand how it actually is!

1. Each instruction will use exactly 4 cycles:

a) Fetch the instruction from memory and - at the same time - add 4 to the PC

b) Decode and get needed ALU operands and - at the same time in the case of a jump/branch instruction - update the PC to the target address.

To make this work, we'll need two input registers for the ALU to hold the operands for the next step.

c) Perform ALU operation.

To make this work, we'll need an output register for the ALU to hold the result for the next step.

d) One of the following

(1) Store computed result into a register

(2) Read from memory location specified by computed result and put value read into a register

(Note: this combines two steps from the example in the book, since only one type of instruction needs two steps here - so I "fudged" a bit!)

(3) Write a register into memory location specified by the computed result

2. This multi cycle implementation addresses issues with the single cycle implementation we addressed early.

   a) The issue of delayed branch goes away, since branch/jump instructions update the PC on cycle 2 while the next instruction is not read until cycle 1 of the next set. (But this issue will come back when we get to the actual pipelined implementation!)

   b) A multiport memory is no longer needed, since instruction read occurs on cycle 1 and data read/write on cycle 3. (But this issue will also come back when we get to the actual pipelined implementation!)

   c) Since each of the 4 cycles takes about the same amount of time, the overall time for 4 cycles is similar to the time for a single cycle in the single cycle implementation (which had to allow time for all the steps in a single cycle.)

   d) Multiple uses of functional units and loops are not issues with the MIPS ISA. If we used this approach with a CISC, we could re-use functional units on different cycles and could allow instructions to take varying numbers of cycles. (E.g. each loop iteration of a string instruction could be its own cycle.)

3. Let's see how this works for an actual instruction

   Example: Add the contents of registers 5 and 6 and puts the result in register 4:

```
sub $4, $5, $6

000000 00101 00110 00100 00000 000000
0x00a62022
```

DEMO (using Multi-Cycle implementation set to manual control)

a) Load machine language into location 0; initialize $4 to 0, $5 to 1, $6 to 2, and PC to 0

b) Ask class to develop RTL for each step - demo each in turn

```
IR <- M[PC], PC <- PC + 4
ALUInputA <- register[rs], ALUInputB <- register[rt]
ALUOutput <- ALUInputA func ALUInputB
(note how simulation shows function as -)
register[rd]<- ALUOutput
```

D. Handout complete RTL for Multi-Cycle MIPS Implementation; then go through

E. Two more demos. In these cases, we will let the control words be generated automatically using the Control unit which we will discuss next

1. The following adds 1 to the contents of memory cell 0x1000:

```
lw $2, 0x1000($0)
addi $2, $2, 1
sw $2, 0x1000($0)
```

This corresponds to the following machine-language program:

```
8c021000
20420001
ac021000
```

HANDOUT

DEMO: Load program, examine memory location 1000; step through execution using hardwired control.

2. We will now execute the machine language program developed for Part I of Lab 5 on our simulators.

HANDOUT

DEMO: Load, execute with initial value in $4 = 3.

* Note that each program terminates with “dummy instruction”
- b . which produces an infinite loop. In our second example, this would be our jr if we had a main program as in lab.