CPS311 Lecture: CPU Implementation: Data Paths

Last revised August 13, 2021

Objectives:

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

Materials:

1. Projectables
2. smips and mmips demo programs
3. Handout showing RTL for single cycle simulation and for multicycle
4. Handout program that demonstrates different types of instruction
5. Demo file for the above on Single Cycle simulation
6. Circuit Sandbox simulations from CPU Builtins Lecture
7. Additional Circuit Sandbox simulations: General shifter, Comparator

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) x86 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.

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, and multicore computers have several CPU's on 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. (The original designers of MIPs have a written a textbook which discusses this and have made details widely available - many of which are used in our text!)

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 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 memory operations.

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

D. Throughout the course, we have been making use of a fundamental principle in computer science: the notion of levels of abstractions. Recall the levels diagram we looked at early in the course.

Level	Language(s)
HLL Programming	Python, Java, C etc.
Architecture	Machine Language specified by an ISA
MicroArchitecture	RTL
Building Blocks	Devices such as Adders, Registers, MUXes, Memories etc.
Digital Components	Gates, Flip-Flops, Memory Cells
Physical Realization	Electronics, Physics

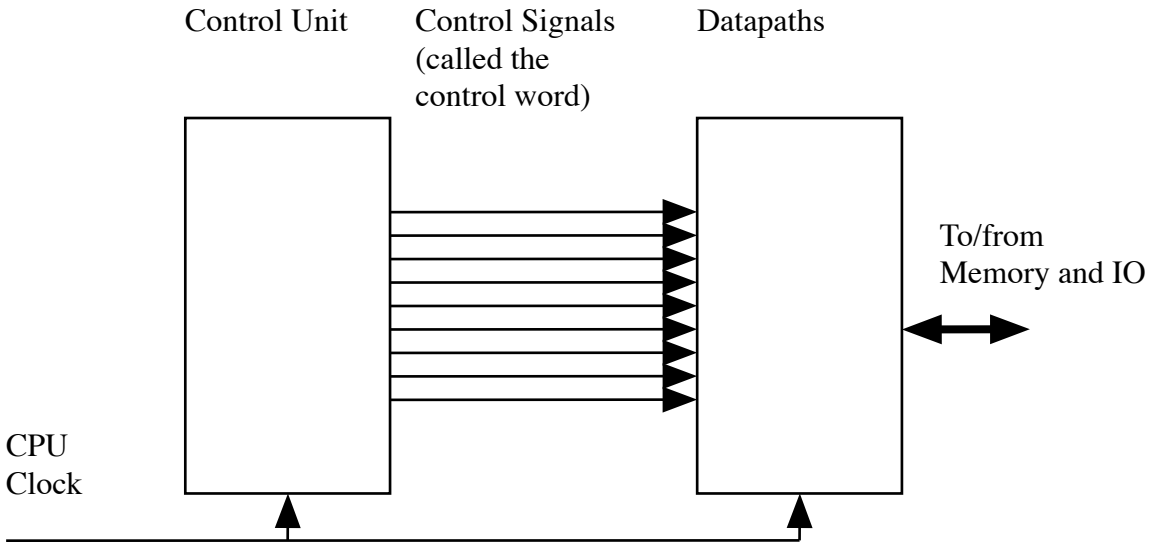
PROJECT

1. Early in the course we looked at Digital Components (Gates etc.) and then saw how they could be combined in various Building Blocks.

2. More recently, we have been lookng at the ISA level.

3. Today, and for several weeks, we look at the MicroArchitecture/RTL level.

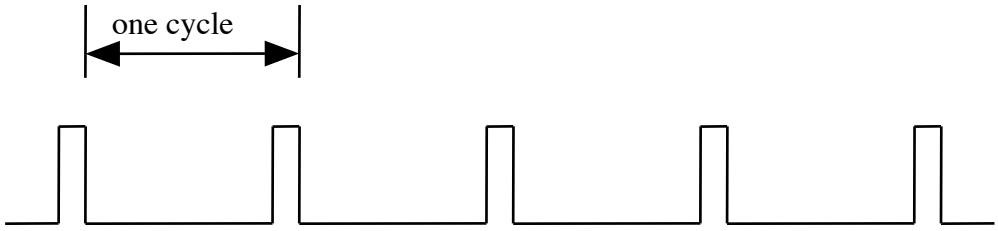
E. A CPU can be regarded as having the following overall structure:



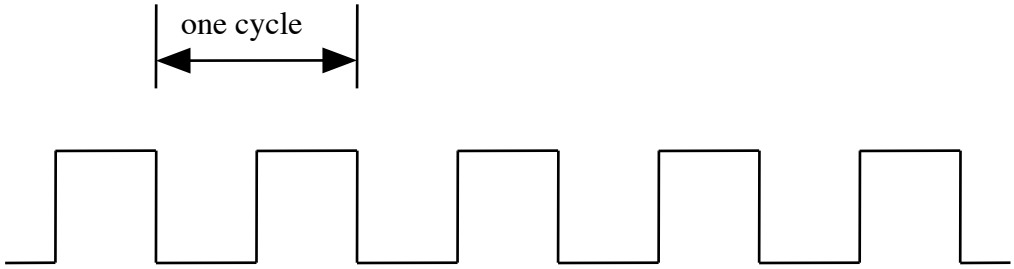
PROJECT

1. The portion on the right (the datapaths) 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. It may also contain other registers needed for the implementation as well. 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:



PROJECT

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.

(1) With a few exceptions we will note later, all the registers receive the same clock signal, but each has a load enable bit in the control word that specifies whether or not that register changes state on the clock.

(2) In the examples we will be developing, we will assume that all state changes take place on the falling edge of the clock. This differs from the discussion in the book, in which state changes take place on the rising edge of the clock. (The motivation for this is that it is consistent with the flip flop chips we have used and will use in alb.)

(3) 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 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.

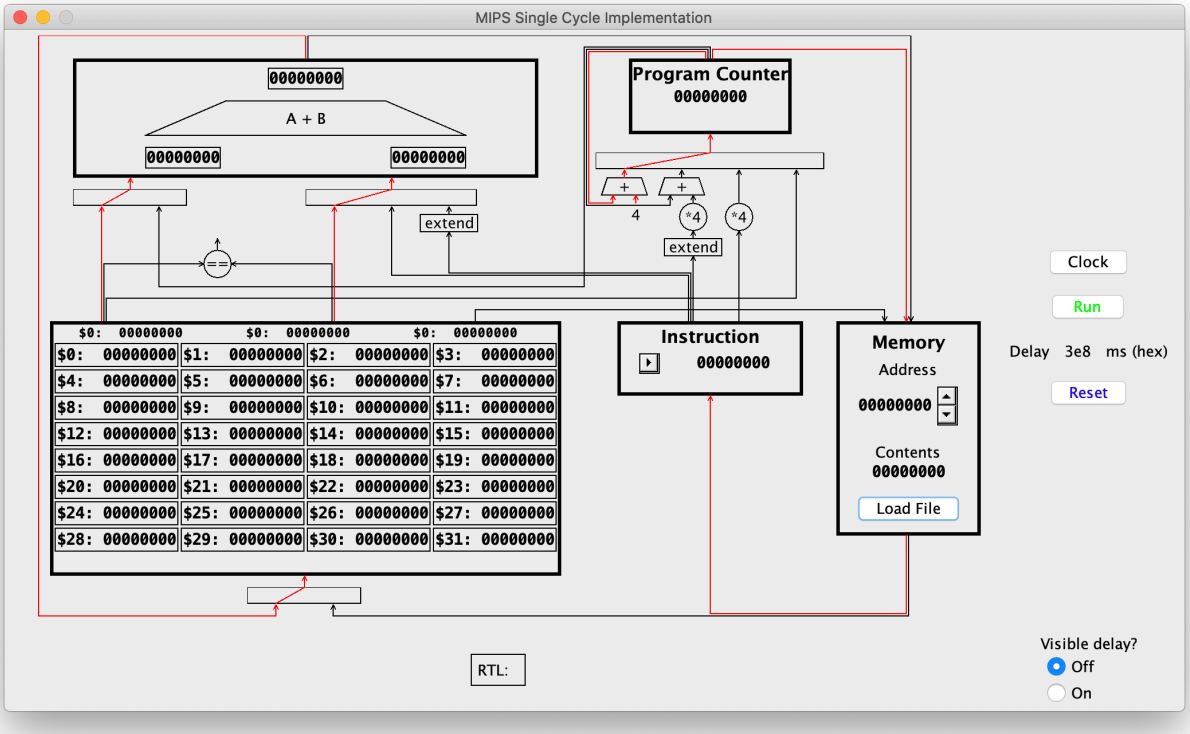
4. It is common today to find CPU's that have a CISC ISA being implemented by a RISC microarchitecture (RISC core) We will not, however, pursue this topic since things can get quite complex!

B. The book discusses two implementations of the MIPS ISA: a single cycle implementation and a multicycle one.

1. We will develop an implementation similar to the first of these, based on a software simulation that will make it possible to observe the internal processes in detail.

2. We will introduce a multicycle implementation later, will discuss how the control unit for it is implemented, and will use it as part of our transition to considering the pipelined implementation which is much closer to the way actual MIPS implementations are structured.

C. The following is a block diagram of the Datapaths for the MIPS simulated implementation we will be discussing today.



PROJECT smips.jar

1. 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 and registers, such as fields that select rs, rt, and rd registers and constants used for I and J format instructions.

   a) On each clock, it is loaded with an instruction to be executed.

   b) During the interval between clock signals, it holds the instruction that is being executed.

   c) On the next clock, the machine state is updated as required by the instruction.

   d) Also, on this same clock the next instruction to be executed is loaded into it.

2. The Program Counter (PC) is part of the architectural state. It holds the address of the next instruction to be executed. It can be updated in one of four ways as determined by the MUX below it.

   a) Its current value plus 4.

   b) The value of the rightmost 16 bits of the IR - multiplied by 4 and sign extended (used for beq/bne)

   c) The value of the rightmost 26 bits of the IR - multiplied by 4. (used for j, jal)

   d) The value in the register specified by the rs field of the current instruction (used for jr, jalr).

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

   a) It has two outputs (at the top) that go to the ALU - and one also goes to the memory. The outputs carry the values specified by the rs and rt fields of the instruction in the IR.

   b) It has one input (at the bottom) which provides the new value to be loaded into one of the registers in the last step of certain instructions. There is a single load enable for the register set as a whole, which is passed on only to the selected register.

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

   a) It has two input (at the bottom) that contain the values to be operated on. (For some operations only the left value is used - the right is ignored.)

(1) A MUX determines whether the left input value comes from a register in the register set (the one specified by rs) or the PC.

(2) Another MUX determines whether the right input value comes from a register in the register set (the one specified by rt) or from the immediate value field of the current instruction - or the immediate value sign-extended.

b) The operation it performs is specified by bits in the control word and/or the funct field of the instruction currently being executed,

c) It has one output. After a delay for signal propagation through the various gates in the input MUXes and ALU, it will contain the result of applying the specified operation to the input(s).

5. The Memory, though not part of the CPU per se, is shown because data flows to/from it from the portion we are focussing on.

a) It needs to be able to do two different things at the same time:

(1) Read an instruction from a memory address specified by the PC.

(2) Read or write a data item at an address computed by adding a register and a 16 bit immediate value that is part of an lw or sw instruction.

b) To accomplish this, it has two distinct ports, each of which has address and data lines.

(1) Its instruction port receives an address from the PC, and sends data to the IR.

(2) Its data port receives a computed address from the ALU. It can either send data to the register set (for lw), or receive data from the register set (for sw).

6. One comparator is used to compare the rs and rt registers in the register set for equality. This comparator produces a result of 1 if the registers are equal and 0 if they are not.

7. 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.

8. 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.

III.RTL

A. The operations performed in the Datapaths to execute the various instructions in the ISA are specified by a notation known as Register Transfer Language (RTL).

   DISTRIBUTE RTL example program using all instructions For now, we will look only at single cycle version on page 1

B. DEMO using smips.jar

   1. Load sample program into memory

   2. Set initial values in registers to r1 = 1, r2 = 2, r3 = 3

3. Open disclosure triangle on IR.

4. Walk through program instruction by instruction, showing correlation between data paths and RTL

5. For add, addi note how destination register changes on clock at end

6. For sw note how memory location 1000 has been updated after clock

7. For lw note how destination register changes on clock at end

8. For beq

    a) Note how register outputs are set up to compare

    b) Note how datapaths are set up to compute target address as PC + 4 * immediate (in PC adder, not ALU)

    c) Note that PC will not be updated until clock - at same time instruction is fetched - so the instruction fetched will <u>not</u> be the branch target - it will be the physically next instruction.

9. For nop

    a) Note how op-code 0 is actually an R-Type instruction that stores a result in $0 - hence it is vacuous.

    b) Note how PC is the branch target from the previous beq

10. Note how bne is not done since registers actually are equal

11. For j

    a) Note how datapaths are set up to compute target address as 4 * immediate

b) Note that PC will not be updated until clock - at same time instruction is fetched - so the instruction fetched will not be the jump target - it will be the physically next instruction.

12. For jal

a) Note how datapaths are set up to compute target address as 4 * immediate

b) Note that PC will not be updated until clock - at same time instruction is fetched - so the instruction fetched will not be the jump target - it will be the physically next instruction.

c) Note how $31 contains the address of the instruction just after the jal

13. Note infinite loop at end involving bne and nop.

IV. Building the Datapaths from Building Blocks at next level down

A. Review building blocks from earlier lecture

DEMO and review CircuitSandbox implementation of each

1. Adders
2. Decoders
3. Multiplexers
4. Registers
5. Shifters

B. 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 copies of the register bit circuit - all sharing a common enable (and clear if needed).

2. The data in bits connect to the data path going into the register, and data out to the data path going out of the register (each 32 bits wide in the case of mips).

3. As we have already noted, all the register flip flops (over 1000 of them in mips) are connected to the clock. 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. Since 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.

   Review CSB Register Demo

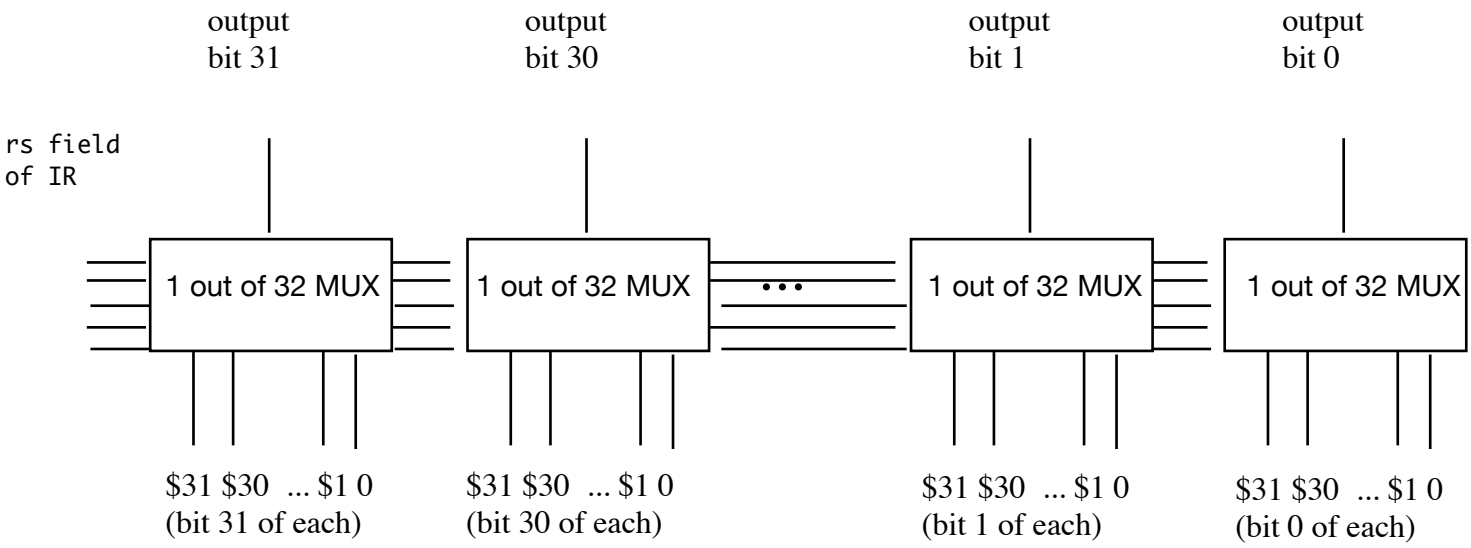
C. Implementation of the IR is straightforward - it is just a simple register, with the inputs connected to the memory, the outputs controlling various functions - and going to the control unit - and the load enable is always 1 (which, in this simulation, is actually done at the end of executing the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

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.

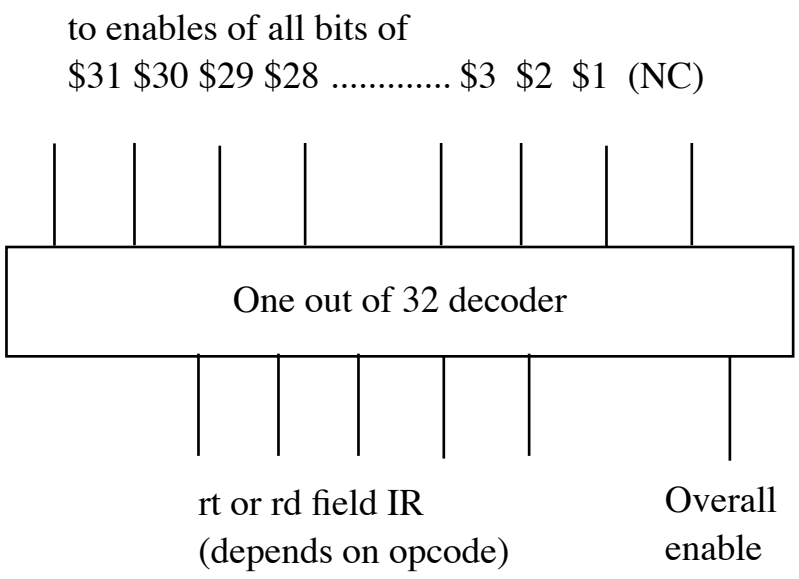
   For example, the left most (rs) output may look like this



PROJECT

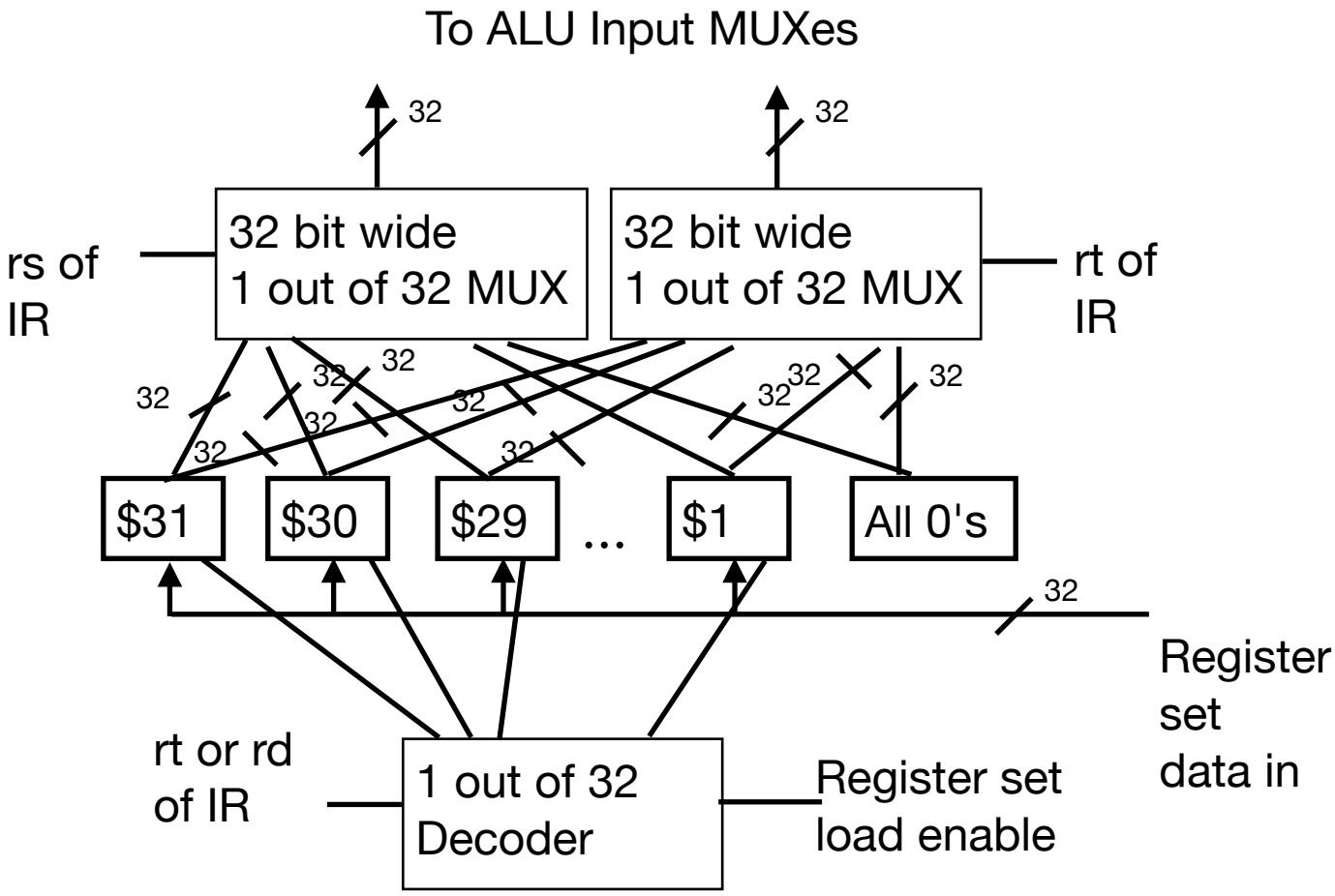
3. The corresponding data inputs of each register may be connected to the data input to the register set - e.g. bit 31 of each of the registers ($1..$31) may all be connected to data input bit 31, etc.

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!



PROJECT

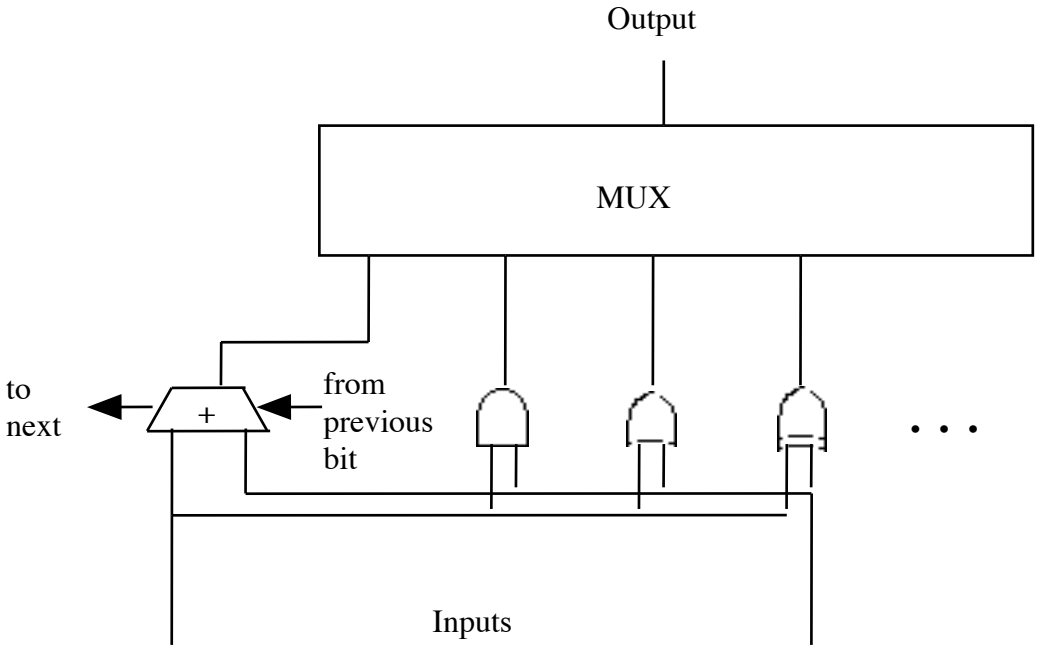
E. The ALU of MIPS performs one of 10 different operations on two 32 bit input values to produce a 32 bit result - with some additional variations for the shifts. It might implemented by 32 copies of a circuit consisting of a MUX plus appropriate gates/gate networks for each function.

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

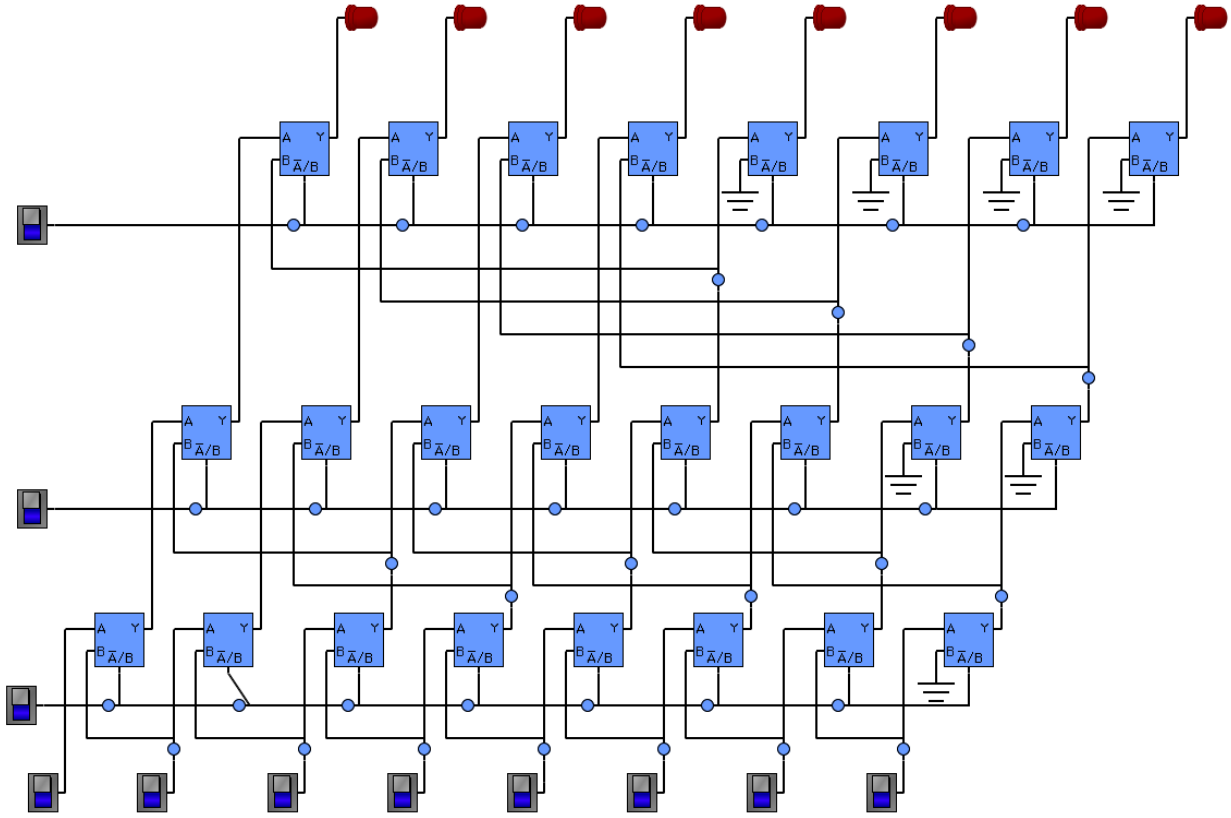
   Output ← A + B
   Output ← 1 if A < B (slt)
   Output ← A & B
   Output ← A | B
   Output ← A ^ B
   Output ← B << 16
   Output ← A (B ignored)
   Output controlled by funct field of IR (numerous possibilities)

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

   PROJECT



3. ALU shifters that shift left or right an arbitrarily-specified number of places (to support instructions like shl, shr, ashl, and their "v" versions) can be built using multiplexers.



DEMO CSB General Shifter

a) Observe that it has an 8 bit input through the switches on the bottom and provides an 8 bit output through the LEDs on the top. (It could be built with any number of inputs and outputs - e.g. the ALU of mips contains a 32 bit shifter but producing such a demo would be a lot of work!

b) It has three control inputs on the left. **Discuss details below only if time allows**

(1)If all control inputs are off, the value input value is passed through unchanged to the output.

DEMO

(2)If the first (lower) control input is on, the value input is shifted left one place to the left.

DEMO

(3)If the second (middle) control bit is on, the value input is shifted two places to the left.

DEMO

(4)If the first and second control bits are on, the value input is shifted three places to the left.

DEMO

(5)If the third (upper) control bit is on, the value input is shifted three places to the left.

DEMO

(6)What will happen if the just first and third control bits are on?

ASK

DEMO

(7)What will happen if all three control bits are on?

ASK

DEMO

(8)Observe that if just the first (lower) control bit is on, the input is shifted left one place; if just the second control bit is on, the input is shifted left two places, and if just the third control bit is on, the input is shifted left four places.

What pattern do you see?

ASK

If we added another row of multiplexers following the same pattern, how much of a shift do you think their control bit would cause?

ASK

(9) Observe that, if we numbered the control bits 0, 1, 2, they would correspond to a shift of 2^0 , 2^1 and 2^2 places respectively.

c) Of course, the same approach could be used to build other kinds of shifters.

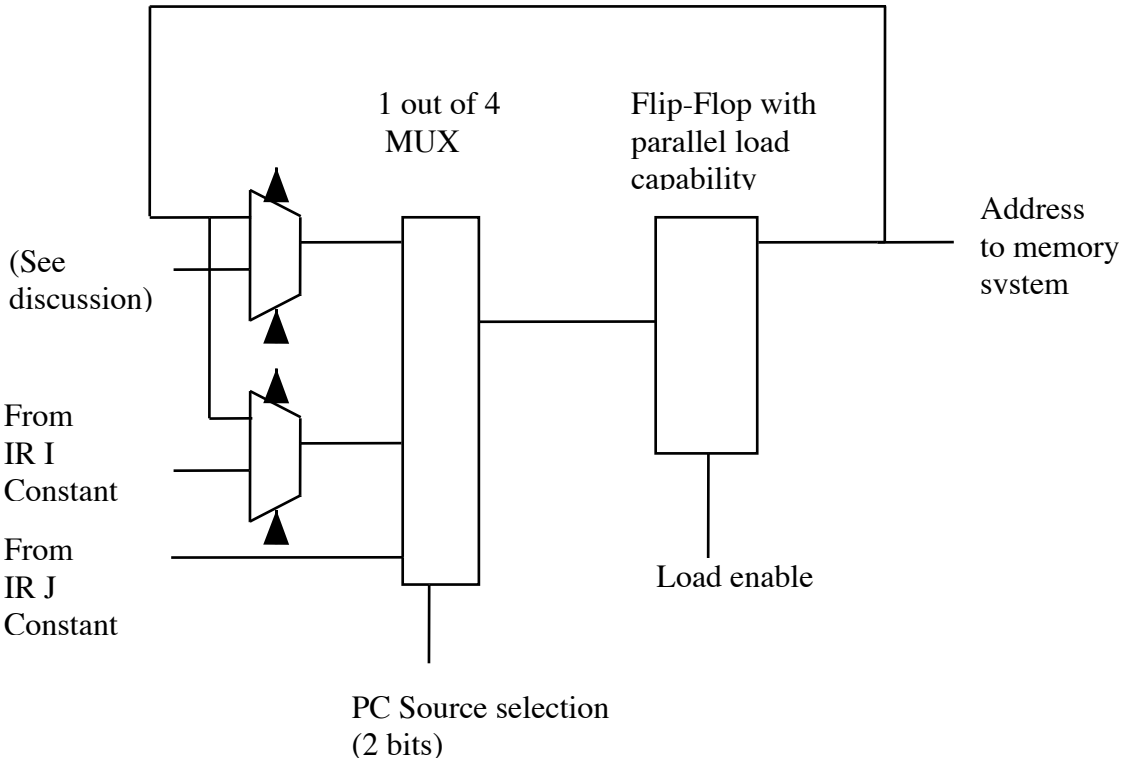
(1) A shifter that shifts any number of places (just more rows of muxes)

(2) A shifter that shifts right instead.

(3) A shifter that does arithmetic right shifts (shift a copy of the sign, rather than 0, into the leftmost bit(s)).

d) We could also use 4 way multiplexers with two selection bits to implement the shifter with half the multiplexers.

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



PROJECT

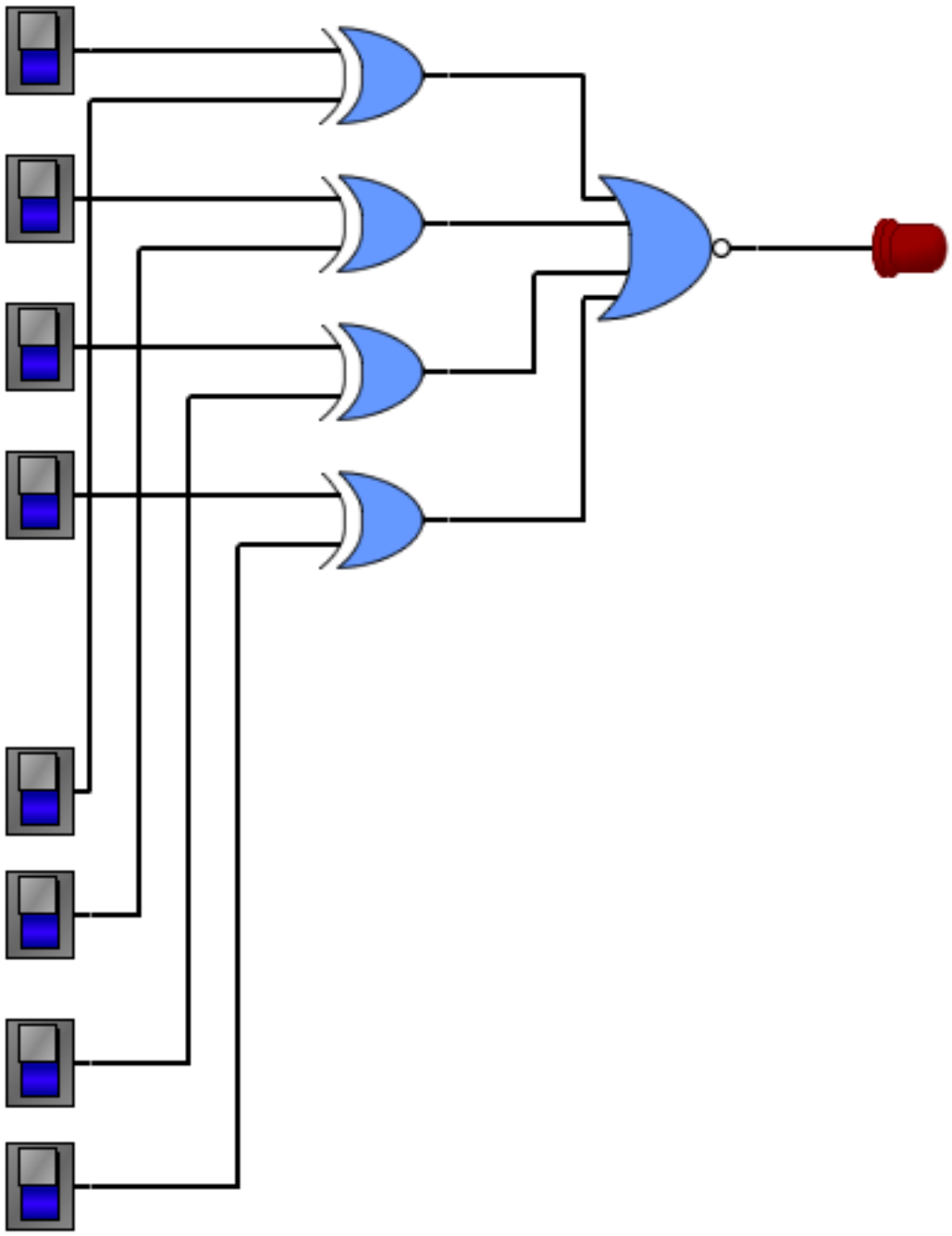
1. Adding + 4 is achieved by hard-wiring the second input of bit 2 of the adder to be 1, and all others to be 0.

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.

G. The rs/rt comparator for equality could be implemented by 32 xor gates that compare each pair of bits and a nor gate which outputs a 0 just when any pair of corresponding bits differ.



DEMO: CSB Comparator

V. 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: After it is fetched, an lw instruction performs the following operations, each of which depends on the result of the one just before.

(1) Add register specified by IR and constant in IR in ALU

(2) Fetch data from memory from address specified by ALU output

(3) Store value fetched into a register

DEMO: smips simlation

(a) Put put `8c021000` into memory location 0

(b) Reset

(c) Open disclosure triangle on IR

(d) Enable injection of delay and set delay to 1388 (=5 ms)

(e) Clock and note steps

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 at most 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.

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) is not much more than the single cycle time for short instructions, and can easily vary from instruction to instruction so that some instructions complete in fewer cycles than others.

   (Actually, for MIPS doing this will actually make execution time for an instruction worse - but this is a precursor to a strategy known as pipelining we will discuss in a few days, which is several times faster.)

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

D. Refer to page 2 of RTL handout - multicycle implmentation; then go through same program as done earlier. Note that each instruction executes in four clocks.

E. DEMO with mmips.jar.

1. Set for hardwired control

2. Put 1 in $1, 2 in $2, and 3 in $3

3. Use most instructions program.

4. Note that ALU input and output registers are updated on clock at end of cycle!

5. Note that no need for delayed branch/jump - this has gone away for now, but will return when we get to pipelining!

VI. Controlling the Operation of the Data Paths - COVER AT START OF CONTROL UNIT LECTURE IF NECESSARY

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 17 control bits to control everything - all based on the content of the IR!

B. SWITCH DEMO TO MANUAL CONTROL, AND SHOW EACH

1. Three bits (shown as checkboxes on the simulation's manual input panel) disable/enable the loading of a particular register on a clock pulse.

   a) One bit to control whether a register in the register set is loaded. (Often true on Cycle 3 - but not always)

   b) One bit to control whether the IR is loaded. (True only on Cycle 0)

   c) One bit to control whether the PC is loaded. (True only on Cycle 0, but sometimes also elsewhere)

d) The ALU input and output registers are always updated on every clock - ignored when not needed, so value loaded is irrelevant.

2. Two bits control whether a memory read or write is done (cannot both be true at the same time, of course - but often are both false).

3. One bit to control where the address of a memory location to be read or written comes from (the PC or a value calculated in the ALU).

4. Several bits control <u>what</u> is loaded into a given register.

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

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

   c) 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.

   d) 1 bit to control where a new value to be loaded into a register comes comes from - the output of the ALU or memory.

5. 2 bits to control <u>which</u> register in the register set is loaded (determined by rd field of instruction, determined by the rt field, or register 31 (required for JAL))

6. 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.)