CS311 Lecture: CPU Implementation: The Register Transfer Level, the Register Set, Data Paths, and the ALU

Last revised October 15, 2007

Objectives:

1. To show how a CPU is constructed out of a regiser set/ALU/datapaths and a control word.
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. Java Multicycle MIPS implementation simulation
2. Handout of block diagram for the above
3. Circuit Sandbox simulations
   a. One bit of a register
   b. Four bit register
   c. Four bit register with 2 inputs
4. Projectable versions of the following:
   a. Diagram showing relationship between the control word and datapaths
   b. Implementation of a typical bit of the PC
   c. Implementation of a typical bit of the register set
   d. Implementation of a typical bit of the ALU
5. Handout: RTL for multicycle MIPS implementation
6. File containing machine language code for adding 1 to memory location 1000
7. File containing machine language code from Lab 5 Part I
8. Handout with assembly language and machine language versions for 6 and 7

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, as seen by the assembly 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 architecture.

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) Intel IA32 architecture - first used in 80386 family in mid-1980's; still utilized by current Pentiums, and Intel has committed to continuing to develop into the future.

   d) IBM/Motorola PowerPC architecture (the chip used in Macintoshes) - 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.)

B. 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 a hypothetical implementation of a subset of the MIPS ISA.

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

      a) The implementation we will present is a multicycle implementation, in which each machine instruction requires several CPU clocks cycles to execute, and each instruction is completed before the next is begun. The MIPS ISA is designed to facilitate a pipelined implementation in which the execution of several instructions is overlapped in such a way as to cause one instruction to start and one to finish on every clock cycle. (The average time per instruction is one clock cycle, though in fact each instruction still needs multiple clock cycles to execute.)

      b) The implementation 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.

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

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

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

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

c) The implementation we will present does not include some efficiency "tricks".

2. Other ISA's could be implemented using the same basic approach as in this example - though differing significantly in detail, of course. (The same basic overall structure).

3. In a later set of lectures, we will present a pipelined implementation of the MIPS ISA that is similar to what actual implementations look like. Understanding the multicycle implementation we will present here will provide a good foundation for understanding the pipelined implementation. Our goal here is pedagogical.

C. Note that our focus here is on how a CPU is implemented. Later in the course, we will look at memory and IO systems. It turns out that, to a large extent, these are independent of the ISA of the particular CPU they are used with, so we can consider them in isolation.

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. Example: The higher-level language programming level is one way of viewing a computer system, and the assembly-language programming level is the level just below it. A typical statement in an HLL is generally realized by a series of AL statements (though in some cases just one may be required). E.g. in Java or C/C++, we might set the boolean variable `smaller` to `true` or `false` depending on whether the integer variable `x` is less than the integer variable `y`:

```
smaller = x < y;
```

on MIPS, this might be translated into:

```
lw   $4, x
lw   $5, y
slt  $4, $4, $5      # $4 = boolean result of x < y
sw   $4, smaller
```

II. The Register-Transfer level of Abstraction

A. Today, we begin discussing the next level of abstraction down in the hierarchy: the Register-Transfer Level. This level has the following characteristics:

1. The primitive operations at this level are called microoperations

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

3. Each Assembly-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.

B. 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. (See below for discussion of possible destinations and sources)

1. The source may be

   a) A register

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

   c) 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 many 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.

2. The destination may be

   a) A register

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

C. 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)
```

D. 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

E. 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

F. Recall that a CPU consists of two major components.

1. What are they?

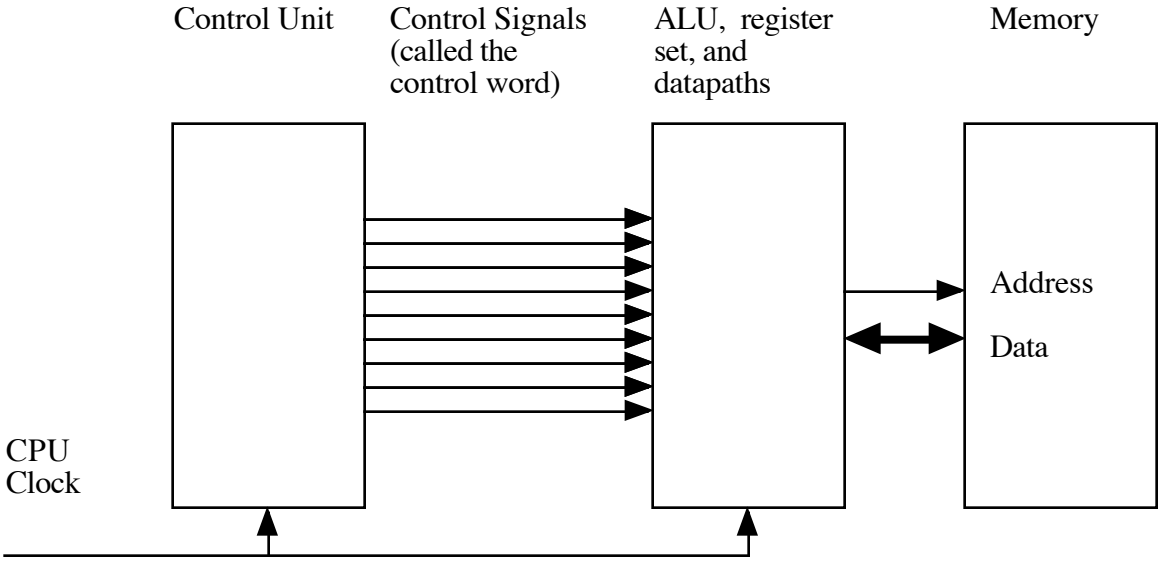
   ASK

   a) The Arithmetic-Logic Unit (ALU), register set, and datapaths (including the path to memory). Sometimes, this is just called the ALU for short (i.e. there is a broad and a narrower meaning of the term.)

   b) The Control Unit

2. It is the task of the ALU, register set, and datapaths to actually perform the various microoperations. It is the task of the Control Unit to sequence the microoperations. That is, they are interconnected as follows:

PROJECT



3. In the rest of this lecture, we focus on the ALU, Register set, and Datapaths. The next series of lectures will deal with the Control Unit.

III. The ALU, Register Set, 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 usually includes registers that do not appear in the ISA.

3. An ISA might have 8 specialized registers, but the microarchitecture might utilize 8 general registers which are mapped to the various special functions in the ISA.

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!

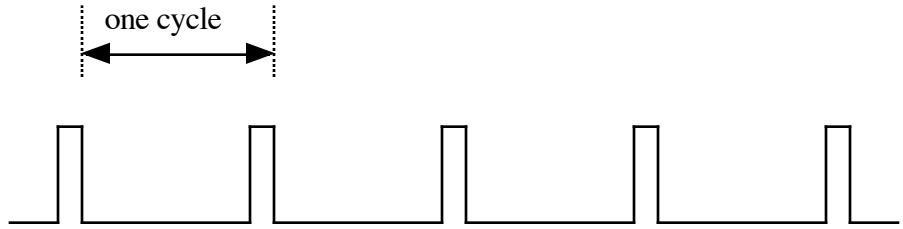
B. Micro-architecture for a Multicycle MIPS Implementation

PROJECT Java simulation

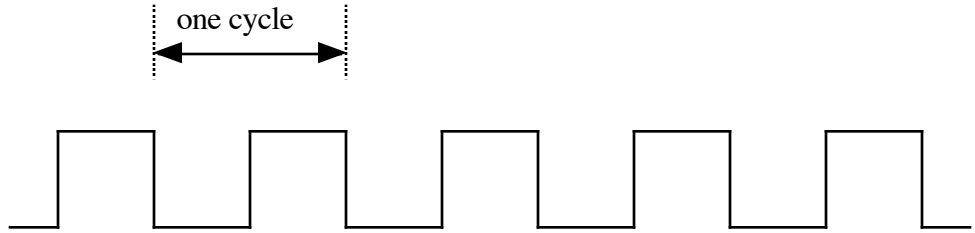
HANDOUT printed version of the diagram

C. The System Clock

1. At the heart of most computer systems is a system clock, whose output looks like this:



or perhaps this:



2. The frequency of the clock dictates the overall speed of the system.

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

   b) 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.

   c) 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”.

3. 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.

   a) In the example 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.)

   b) 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.)

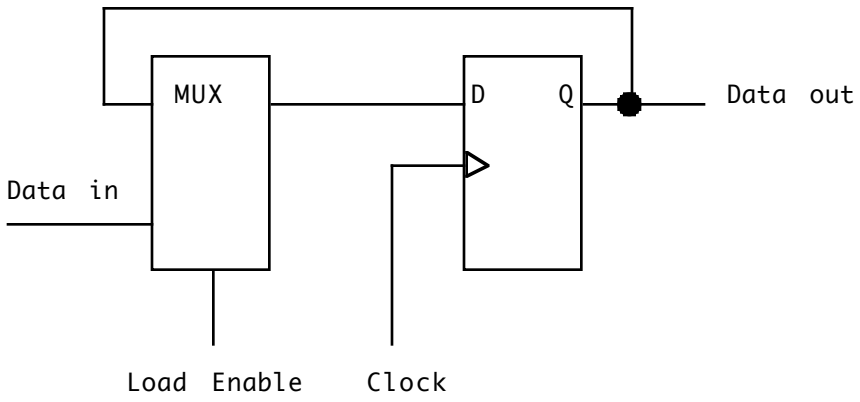
4. In our simulation, the system clock is simulated by the clock button. Pressing it corresponds to a rising clock edge, and releasing it to a falling clock edge.

   DEMONSTRATE WITH CONTROL SET TO HARDWIRED

D. The heart of the CPU is the registers.

1. Registers are typically implemented using flip-flops - one flip-flop per bit.

2. The following is a typical way to implement one bit of a register:



   a) On every clock, the D flip flop loads a new value. Depending on the Load Enable control signal, this is either Q (Load Enable = 0) or the Data in value (Load Enable = 1).

b) Of course, in the former case it would appear to an observer that the flip flop has not changed state - i.e. it has loaded the value that it already contains, so nothing changes.

c) Thus, this device exhibits the following behavior:

(1) When Load Enable = 0, its value does not change

(2) When Load Enable = 1, its value changes on the clok to whatever is on Data input

Demonstrate using Circuit Sandbox simulation.

3. An n-bit register is implemented by using n copies of this circuit. The clocks are all tied together, as are the Load Enables. Each bit has its own Data input and data output.

   Demonstrate using Circuit Sandbox simulation

4. The Load Enable becomes one bit of the control word. Thus, the register behaves as follows:

Load Enable bit in Control Word	Register state after next clock pulse
0	No change
1	Copy of data input (before clock)

E. The registers are connected by various data paths.

1. A microoperation such as

   SomeDestination ← SomeSource

   (where SomeDestination is an n-bit register and SomeSource is an n-bit register or some other n-bit data source) can be implemented by connecting the data inputs of SomeDestination to the data outputs of SomeSource.

2. Typically, though the input to a given register can come from multiple places - e.g. there are multiple microoperations of the form

   SomeDestination ← SomeSource
   SomeDestination ← SomeOtherSource

   In this case, we need selectable data paths

3. One way to implement such data paths is by using MUXes.

   a) Suppose a certain n-bit register can receive input from any one of 2 sources. One way to implement this is with by connecting the data inputs to n 1 out of 2 MUXes, each of which selects one bit of data input from one of the 2 possible sources.

   b) Demonstration: 4 bit register with 2 inputs

   c) The selection input for the MUXes becomes part of the control word, along with the Load Enable for the register. Thus, the register behaves as follows

Load Enable bit in Control Word	Source select bit in Control Word	Register state after next clock pulse
0	-	No change
1	0	Copy of source "A"
1	1	Copy of source "B"

   (When a source is copied, it represents the value of the source as it was before the clock)

   d) Of course, the same approach could be used with more than two sources - e.g. up to four sources are possible by using n 1 out of 4 MUXes with 2 source select bits in the control word, etc.

4. Another way to implement such data paths is by using internal busses. (We won't discuss this option here).

F. Registers and Data Paths in our example

1. The Program Counter (PC) - 32 bits, holding the address of the next instruction to be executed.

   a) An output connected to the memory system

   b) An input which can be connected either to an adder that adds 4 to the current value, or to the ALU output (to support beq/bne), or to the constant field of J Format instructions, scaled by * 4 (j, jal).

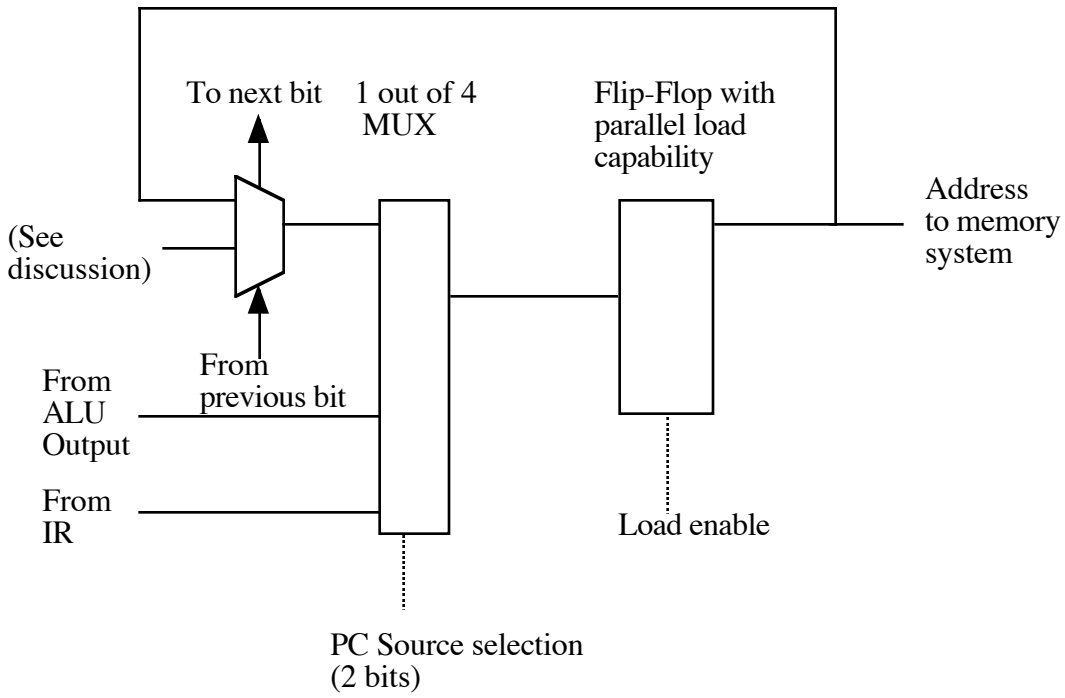
   DEMO Incrementing the PC

   c) The PC is capable of performing the following microoperations:

PC ← PC + 4 - used for all instructions
PC ← ALU Output - used for beq/bne if branch taken
PC ← J Format constant - used for j, jal

d) Implementation: 32 bit register with a parallel load capability and input wired to a 3-way MUX (4 - way MUX with one input unused)

Typical bit PROJECT



e) Notes:

(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) The Constant from the ALU is multiplied by 4, which 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. 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) 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.

DEMONSTRATE trying to load 7 into the PC

2. The Instruction register (IR) - 32 bits register holding the current instruction being executed.

   a) The control unit uses the opcode stored in this register as input.

   b) Various bits in this register are also used to select registers in the general register set, as discussed below.

   c) Various bits in this register (the constant portion of I or J Format instructions) can be sent to the ALU or the Program Counter.

   d) The input to this register comes from the memory. It is loaded when a new instruction is fetched from memory.

   e) The IR is capable of performing the microoperation

      IR ← Memory[Address] - used for all instructions

      DEMO putting a value into M[0] and then loading into IR

   f) Implementation: 32 bit register with parallel load capability - input coming from the memory system

3. The General Register set - 32 registers, 32 bits each

   a) Two outputs. One is the register selected by the rs field of the instruction register contents; the other the register selected by the rt field of the instruction register contents. (If the instruction doesn't have these fields, then whatever registers are selected by the corresponding bits of the instruction are selected, though they may be ignored.)

   b) One input which can be connected to either the ALU or the memory

   c) The register set is capable of performing the following microoperations (where register[rs] denotes the register selected by the rs field of the IR, etc.)

      Register[rd] ← ALU Output - used for RType instructions
      Register[rd] ← Memory[Address] - (not actually used)
      Register[rt] ← ALU Output - used for immediate instr

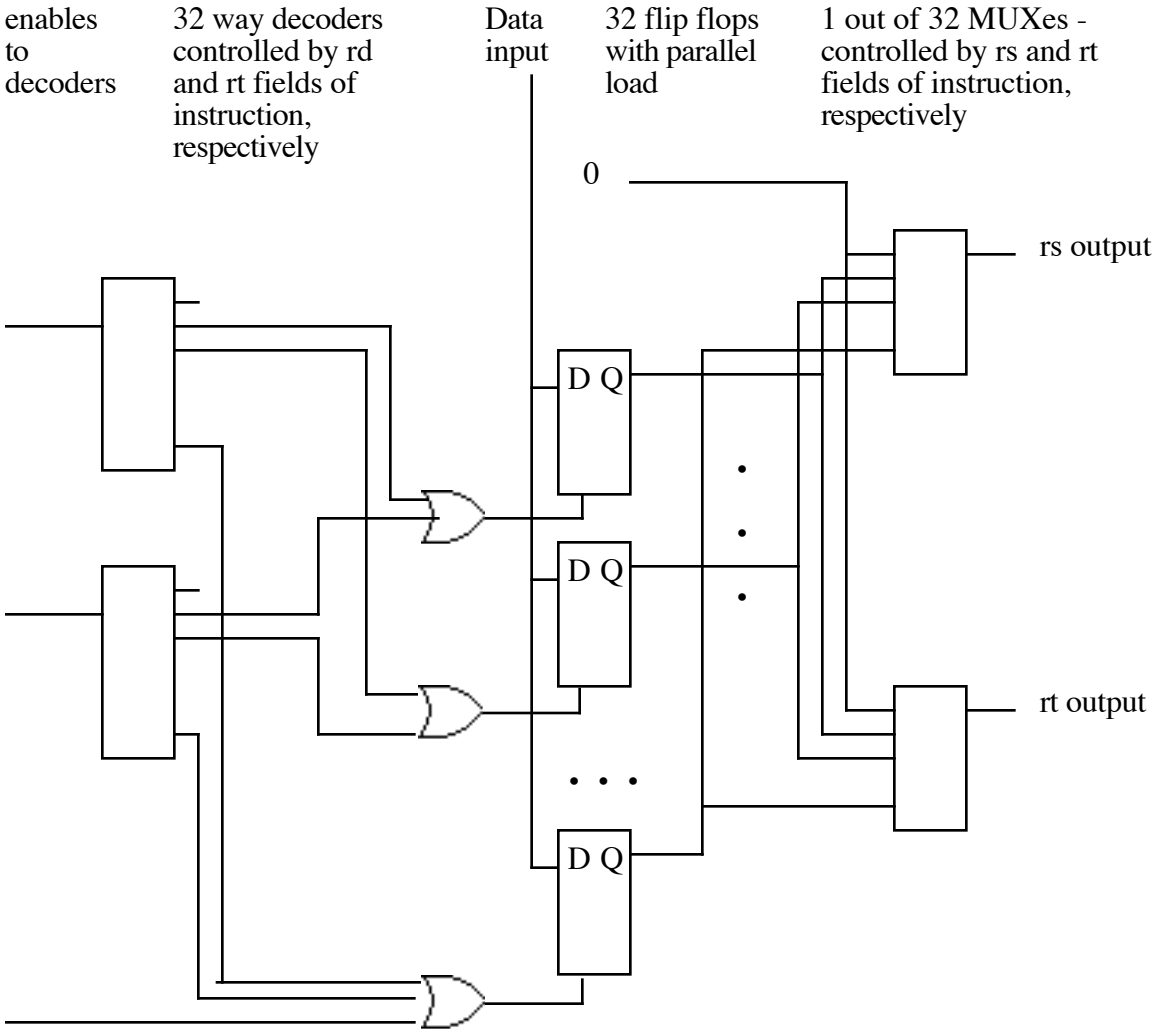
Register[rt] ← Memory[Address] - used for load instructions
Register[31] ← ALU Output - used for jal instructions
Register[31] ← Memory[Address] - (not actually used)

(Note that the implementation makes some operations possible that are never actually used.)

DEMO loading a value into a register from the ALU

d) Implementation: an array of 32 registers - each 32 bit with parallel load capability - plus two arrays of out of 32 MUXes to derive the outputs, and one array of 1 out of 2 MUXes to derive the input, and with three decoders used to derive the parallel load control signals. (Note: $0 is implemented by hard-wiring to 0 and ignoring store attempts)

Typical bit PROJECT



G. The ALU in our example

1. The ALU is a subsystem composed of two input registers, an output register and a set of combinatorial circuits.

   a) There are two inputs, representing input into the two input registers. On every clock pulse, the A and B input registers are loaded from these inputs.

      DEMO: Put 00220000 in IR; expand to show rs = 1, rt = 2, load r1 and r2 with known values and then show effect of clock.

   b) On every clock pulse, the combinatorial circuits compute some function of the inputs, which is then loaded into the output register.

      DEMO

   c) There is one output, coming from the output register.

2. The possible functions the ALU can perform include all the operations needed to perform the various R-Type and immediate instructions. In particular, the combinatorial circuits in the ALU can perform the following microoperations:

   Output ← A + B
   Output ← A - B
   Output ← A & B
   Output ← A | B
   Output ← A ^ B
   Output ← A nor B
   Output ← B << shamt
   Output ← B >> shamt
   Output ← B >>> shamt
   Output ← B << A
   Output ← B >> A
   Output ← B >>> A
   Output ← A < B
   Output ← B << 16
   Output ← A

3. The specific computation performed by the combinatorial circuits is either specified explicitly or is determined by the func field of the current instruction (for R Type instructions)

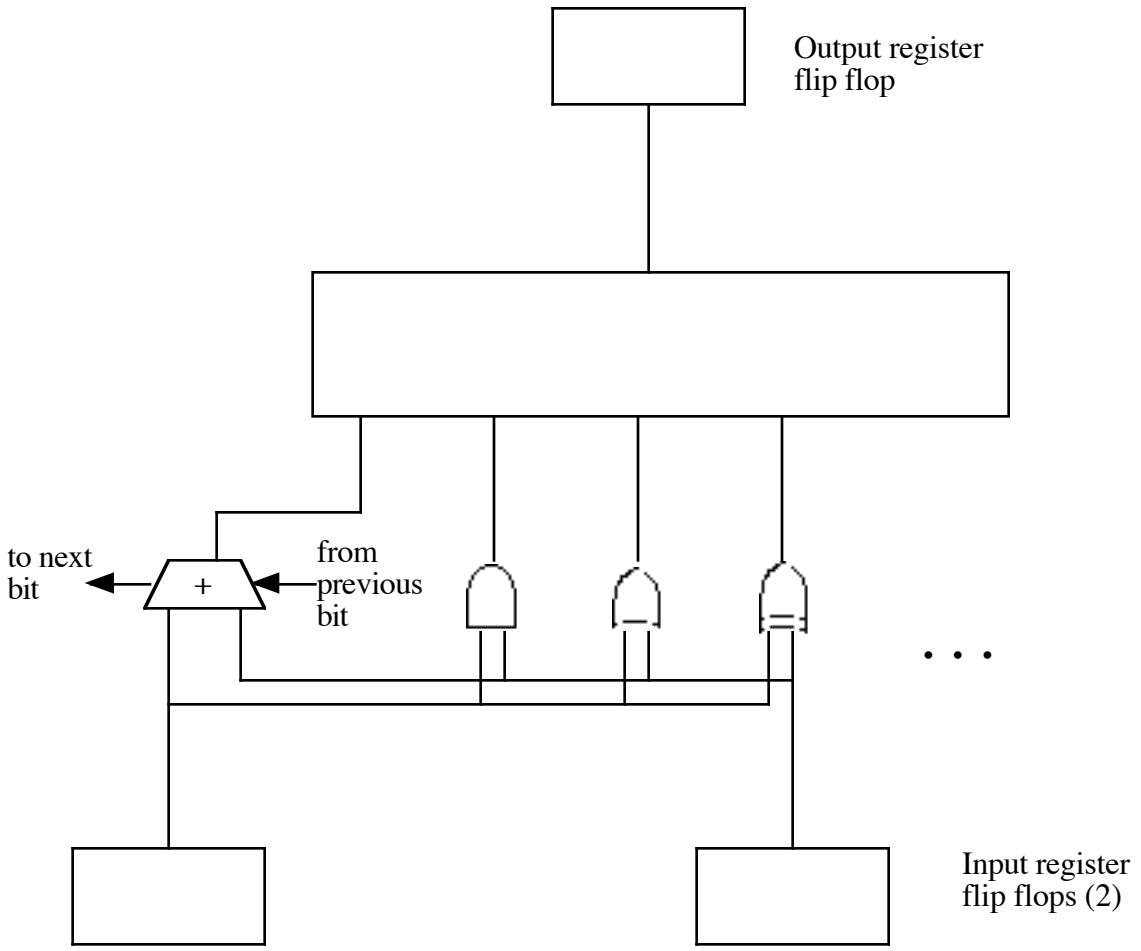
   DEMO various operations

4. Implementation

a) Three registers: three arrays of flip flops that are always loaded on every clock.

b) Combinatorial network, with each function implemented and one function selected by a MUX

Typical Bit PROJECT



H. The Memory System

1. The memory system is actually a distinct portion of the overall computer system, which has a great deal of internal complexity of its own. We will discuss the structure of the memory system later in the course. For now, we think of the memory as a "black box" whose interface is what is shown in the diagram.'

2. The memory has two inputs - an address and data - and one output. The address in can come either from the PC or the ALU output; data in comes from the register set, the output can go either to the register set or the IR. It can perform the following microoperations (among others).

   IR ← Memory[PC] (Instruction READ) - Data in is ignored
   Some register ← Memory[ALU out] (Data READ) - Data in ignored
   Memory[ALU out] ← Data in (WRITE) - Data out is ignored

   DEMONSTRATE

I. Data Paths

1. The data paths provide for the flow of 32-bit words of information between the other components, as necessary to support the microoperations needed to implement the ISA.

2. The data paths consist of wires plus MUXes used to allow multiple inputs to a single component. These are under the control of bits in the control word.

   DEMONSTRATE

IV. Executing Machine-Language Instructions as Series' of Microoperations

A. Each instruction in the ISA is realized by a series of microoperations, which can be expressed as an RTL "program" for carrying it out.

B. To see how individual machine instructions can be executed as a series of microoperations, let's walk through the execution of a single instruction, which adds 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
```

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

2. 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
```

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

D. 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 .-4 which produces an infinite loop. In our second example, this would be our jr if we had a main program as in lab.