University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

January 11, 2001

Computer Organization

Slide 1–1

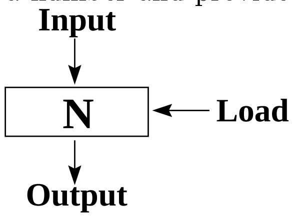
Let's Build a Computer!

- The best way to understand why computers are built the way they are is to build one. Rather than motivate computer design from the top-down (e.g. why doesn't the computer's electronics understand English?), I'll motivate it from the bottom-up (e.g what can we do with the basic electronic building blocks?)
- Suppose we want to build a simple computer that can compute a running sum. To do this we'll need some components from our friends the electrical engineers.

Slide 1–2

Component #1: Register

- A device that can store a number and provide it on request.

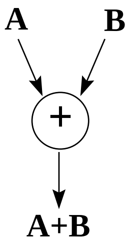


- A register stores the number on its input wires when the button attached to the Load wire is pressed. The value currently stored in the register ('N') is always produced on its output wire.
- When the power is turned on, the register contains '0'.

Slide 1–3

Component #2: Adder

- A device (adder) that can add two numbers:

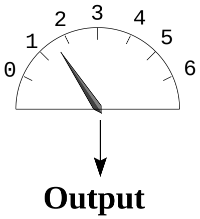


- The adder continuously takes the two numbers on its inputs, computes the sum, and produces the sum on the output.
- If A or B changes, so does the sum.

Slide 1–4

Component #3: Input unit

- A dial that produces on its output the number that is dialed in by the user:

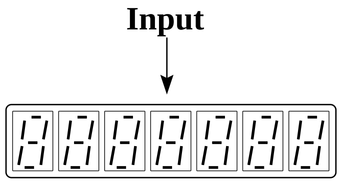


- Assume the dial is big enough to handle any numbers that need to be added.

Slide 1–5

Component #4: Display unit

- A display that displays the decimal value of its input.

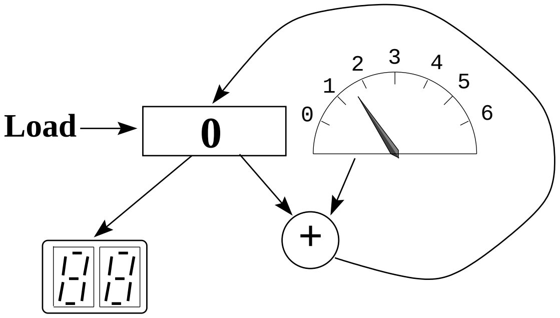


- Assume the display has enough digits to produce the result of any addition.

Slide 1–6

Let's Put it Together!

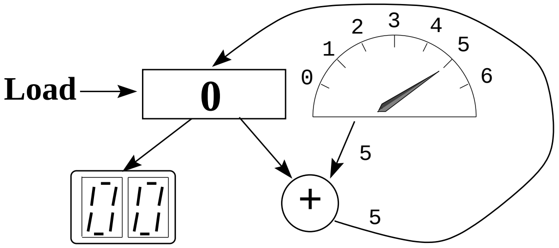
- With these components we can build our simple adding computer:



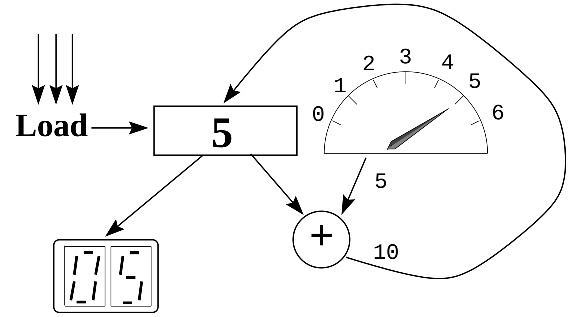
Slide 1–7

Let's Try it Out!

- When the power is turned on the register will contain 0, and the display will show 0. The following happens when adding a number to the sum:
  1. The user sets the dial to the number to be added, e.g. 5.
  2. The register is now producing 0, the dial 5, and the adder therefore 5. The display will still show 0.

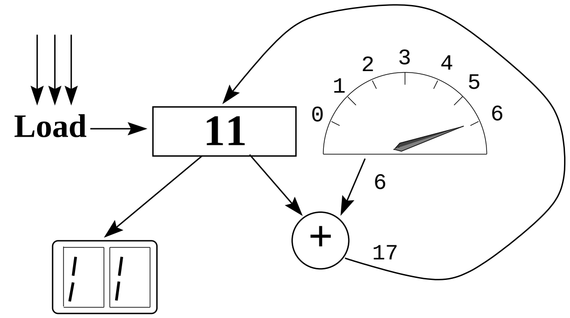


3. The user presses the Load button, causing the register to store 5.



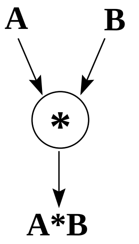
4. The register now produces 5 on its output, so the display shows 5.

5. Note that at this point the adder will be adding $5 + 5$ and producing 10 on its output, but that this value will not be loaded into the register unless the Load button is pressed.
6. To add another number to the sum, the user sets the dial to that number and repeats the process.



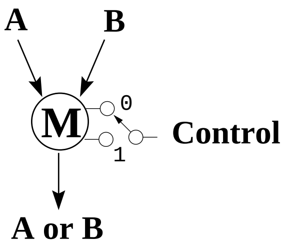
Component #5: Multiply

- We've now built a very simple, but very limited, computer. Suppose we also want our computer to multiply numbers.
- We'll need a device that can multiply two numbers:



Component #6: Multiplexer

- A device that can select between several numbers (a multiplexer).

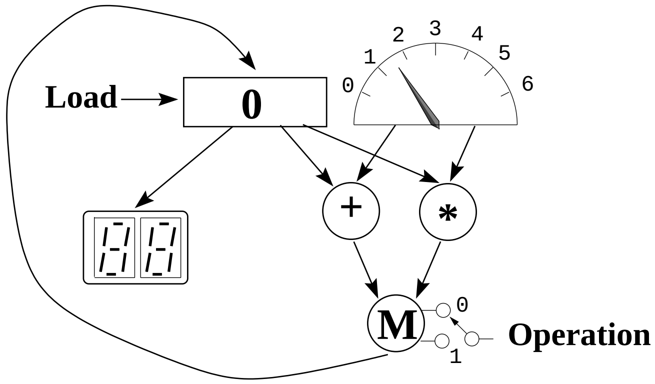


- A multiplexer produces one of N data inputs as its output depending on the value of its control input. If the control is 0 the value of **A** is output, if it's 1 the value of **B** is output, etc.

Slide 1–12

Computer #2: Add and Multiply!

- With these components a computer that adds and multiplies can be constructed:



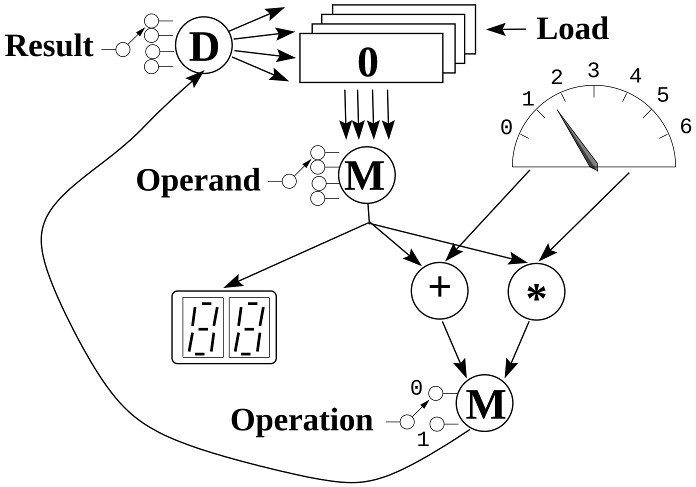
Slide 1–13

Computer #2: · · ·

- The value in the register is both multiplied by and added to the value on the Input dial. The Operation dial selects whether the result of the addition or the result of the multiplication is stored in the register when the Load button is pressed. Assume that 0 on the Operation dial selects addition, while 1 selects multiplication.
- How would you set the register to a particular value?
- Multiplexers also make it easy to support multiple registers, in case we want to save several values, instead of only one.

Slide 1–14

Computer #3: 4 Registers!



Slide 1–15

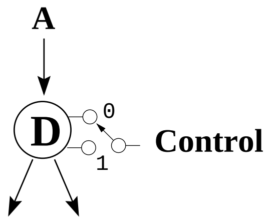
Computer #3: · · ·

- This computer has 4 registers. The collection of registers is called the register file.
- The Operand dial selects which register provides the operand to the operation. The multiple arrows between the register file and the operand multiplexer are meant to indicate one connection from each register to the multiplexer.
- Another Operand dial could also be added for the second operand, allowing it to be provided by a register or the user input.

Slide 1–16

Component #7: Demultiplexer

- The Result dial selects which register stores the result, by determining which register gets the Load signal. This device is a demultiplexer – it decides on which of N outputs the input is produced (it's a multiplexer in reverse).



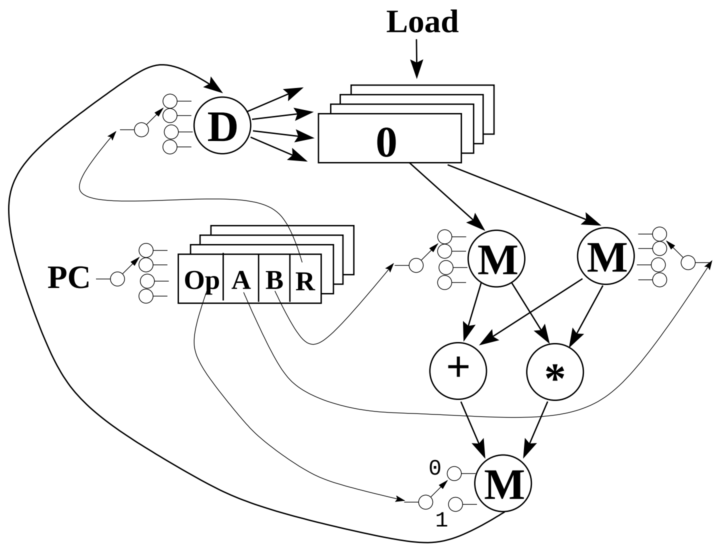
- How would you set a register to a particular value? Can you think of a simpler solution than the 1-register machine?

Slide 1–17

Computer #4: Look Ma, no dials!

- We now have a fairly sophisticated computer. The user can use the Operand dial(s) to select the operands for the operation, the Operation dial to select which operation is performed, and the Result dial to select which register stores the result when the Load button is pressed. The computer runs as fast as the user can change dials and press
- As a final refinement we will dispense with the dials and replace them with registers:

Slide 1–18



Slide 1–19

Computer #4: · · ·

- Each of these new registers contains 4 numbers that control the operation (O), the two operands (I1 & I2) and the result register (R). Each register contains one instruction.
- The encoding of instructions as numbers is called machine code. Machine code is simply numbers that tell the computer what operation to perform.
- The Program Counter dial selects which instruction is to be performed next. The user can make the computer run through the instructions in sequence by alternately incrementing the Program Counter and pressing the Load button.

Slide 1–20

Computer #4: · · ·

- Instructions must be loaded into the instruction registers by the user. An input dial, multiplexer, and load button are needed (not shown) to dial in the instructions and load them into the correct instruction registers.
- Control paths, or connections that control the operation of the computer, are drawn in thin lines. Data paths contain the data being operated upon, and are drawn in thick lines.

Slide 1–21

Computer #5: Clock and PC!

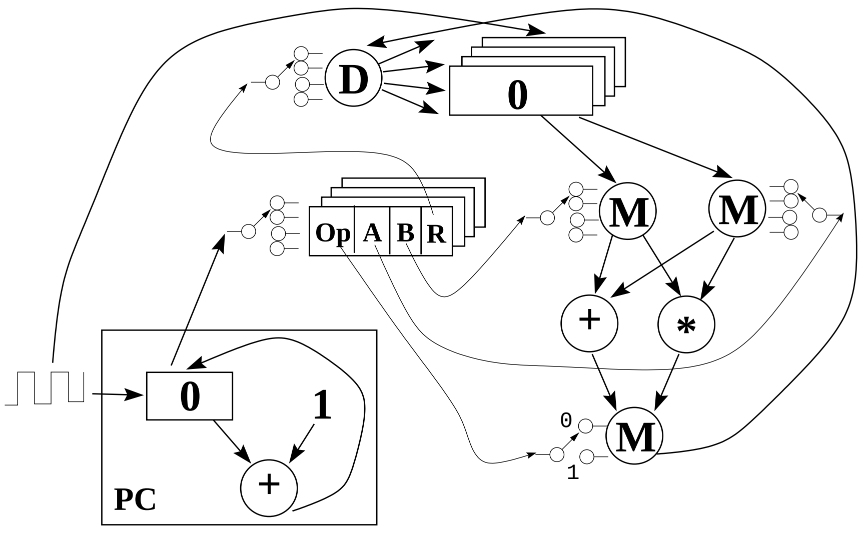
- The final step in making a ”real” computer is to get rid of the Program Counter dial and the Load button. To do so, we need a clock that generates a signal at regular intervals. Each tick of the clock is called a cycle, and in modern computers is measured in MHz.
- The Program Counter dial is replaced by a register and an adder, connected in such a way that each tick of the clock increments the value of the register by one. The value of the program counter (PC) register determines the current instruction.

Slide 1–22

Computer #5: · · ·

- Under normal operation the program counter will increment on each tick of the clock, causing the computer to perform the instructions one after the other.

Slide 1–23



Slide 1–24

Computer #5: · · ·

- The register file is set up to load the result on each tick of the clock.
- That's all there is to a computer. Obviously this description has been simplified, but it's basically correct. One significant "white lie" is that not all instructions can be performed in a single clock tick. Some, such as multiplication, may take several ticks. Thus loading the result into the register file is not done on every clock tick; instead, it depends on the instruction. Also, modern computers can also do more than one instruction per clock tick.

Slide 1–25

Program #1

- We can now write a simple program for our machine:

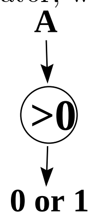
PC	Op	A	B	R
0	*	0	0	2
1	*	1	1	3
2	+	2	3	2

- If register 0 holds the value of x and register 1 holds the value of y, then this program computes $x^2 + y^2$ and stores it into register 2.

Slide 1–26

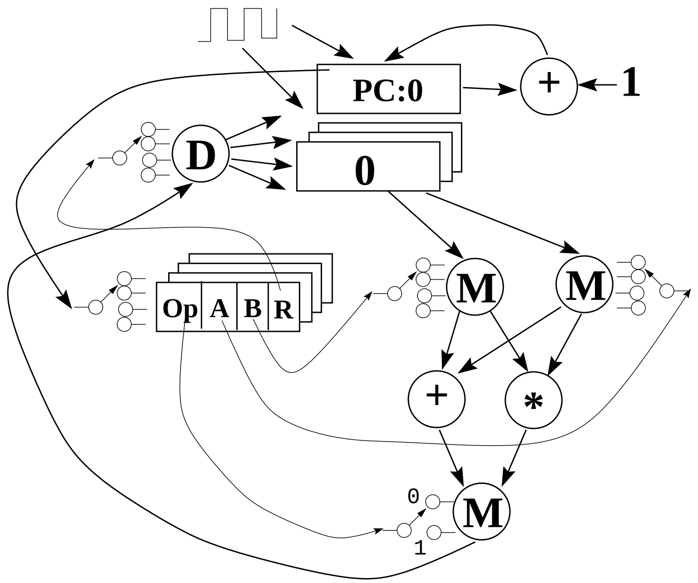
Computer #6: PC as Register

- Because the PC is a register, you can add to it, store to it, etc. For example, you can make the computer jump forward by 10 instructions by adding 10 to the current PC.
- A new component, a comparator, will do this for us:



If `A>0` then the comparator will return **1**, otherwise it will return 0. This allows us to change which will be the next PC.

Slide 1–27



Slide 1–28

Program #2

- A program that computes the maximum of two numbers:

PC	Op	A	B	R
0	-	0	1	2
1	>0	2		2
2	+	PC	2	PC
3	$\rightarrow$	1		4
4	$\rightarrow$	0		4

- Register 0 holds the value of x, register 1 the value of $y \Rightarrow$ the program stores $\max(x, y)$ into register 4.
- '-' does subtraction, $\rightarrow$ moves values between registers.

Slide 1–29

Program #2 (Ooops!)

- Actually, I lied. The program is buggy. (Why?) Here's a corrected version:

PC	Op	A	B	R
0	-	0	1	2
1	>0	2		2
2	+	PC	2	PC
3	$\rightarrow$	1		4
3'	+	PC	#1	PC
4	$\rightarrow$	0		4
5	$\cdots$	$\cdots$	$\cdots$	$\cdots$

- ⌜#1⌝ means the *literal* value 1, as opposed to ⌜1⌝ which means register number 1.

Slide 1–30

Computer #7: Back to Reality!

- The components as described, with the exception of the registers that hold the instructions, are typically found in the central processing unit (CPU) of the computer. The basic components in the CPU are:
- The register file that contains the registers. Typical CPUs have 6- 32 registers.
- The arithmetic logic unit (ALU) that performs arithmetic operations, such as addition, multiplication, etc.
- The control logic that contains the PC and logic for decoding instructions and selecting the correct operation, operands, loading the result, etc.

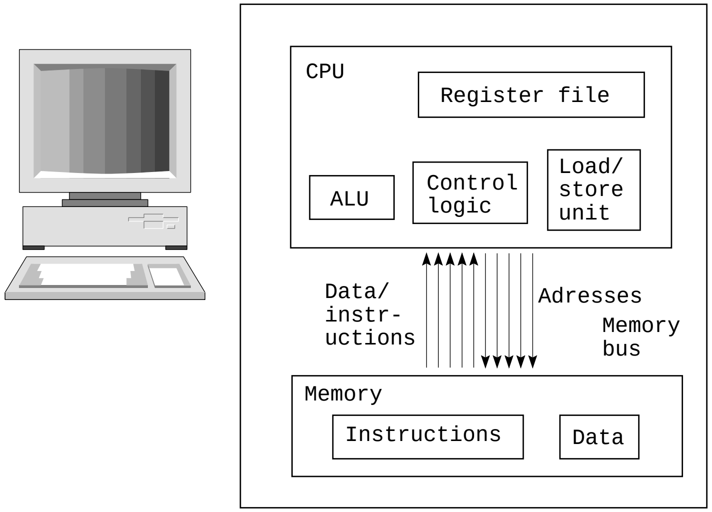
Slide 1–31

Computer #7: $\cdots$

- Fancier CPUs are more complicated and may have several ALUs, caches, floating-point units, graphics accelerators, etc.
- Instructions are not stored in the CPU because there are too many of them. This is also true for the data, because there usually aren't enough registers to hold them all (not enough registers fit on a chip). For this reason a computer has a memory system, which is like a lot of slow registers that exist outside the CPU. The only operations memory supports are reading and writing each individual location (register)

Slide 1–32

Computer #7: · · ·



Slide 1–33

Thanks for the Memory

- The memory and CPU are connected by a bunch of wires called the memory bus.
- Each location in the memory is accessed via its index within the memory, or its address. Each address is unique, starting at 0.
- To read a particular location the CPU sends the appropriate address to the memory on the memory bus, and the memory returns the data at that location to the CPU.

Slide 1–34

Memory · · ·

- To write a particular location the CPU sends the address then the data on the memory bus, and the memory stores the data at the location indicated by the address.
- The CPU can access a register in a single cycle, but accessing memory may take tens or hundreds of cycles. Accessing memory is necessary, but slow.

Slide 1–35

The Fetch/Execute Cycle

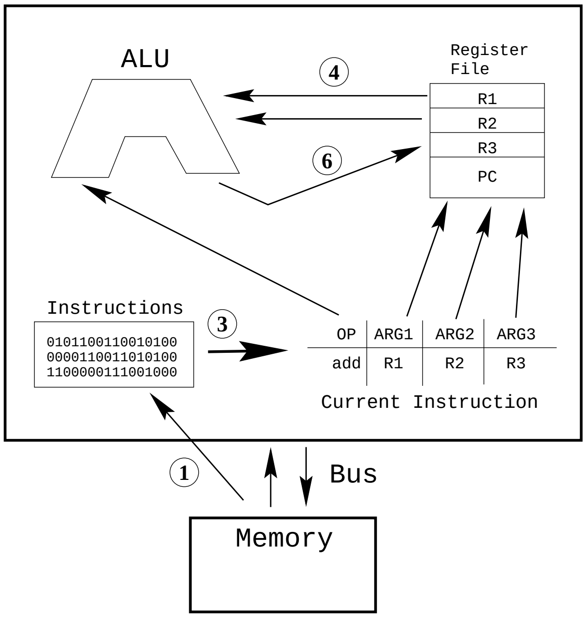
- The PC contains the memory address of the instruction to execute. The CPU must fetch the current instruction before it can execute it. A later topic will cover how to avoid waiting for the memory on every cycle.

The Fetch/Execute Cycle

- The CPU performs the following instruction fetch/execute cycle indefinitely:
  1. Fetch instruction at address stored in PC.
  2. Update PC by adding length of current instruction
  3. Decode the instruction (determine what operation, operands, etc.)
  4. Load the operands into the ALU
  5. Wait for the ALU to perform the operation
  6. Store the result back into the register file (or perhaps the PC)
  7. Repeat

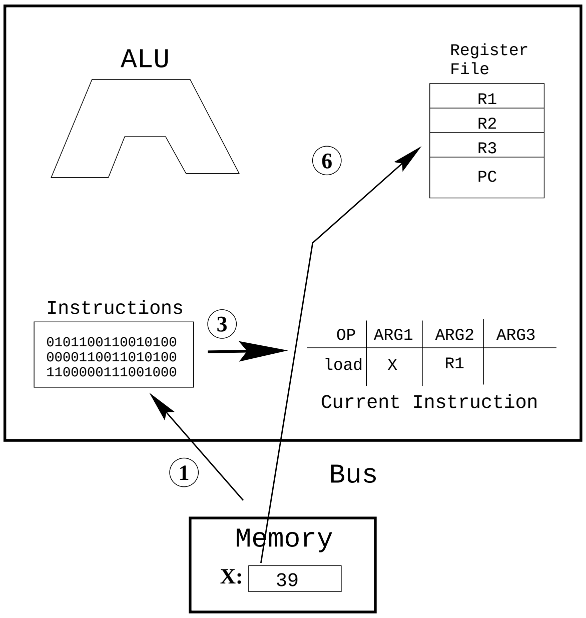
Slide 1–36

Fetch/Execute — Add



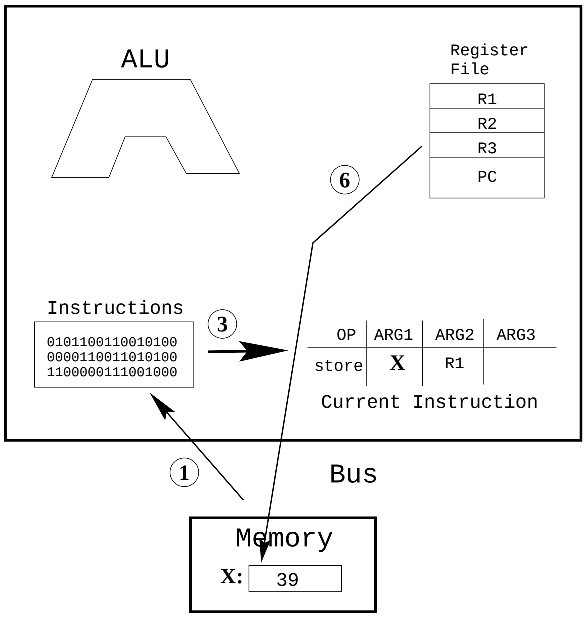
Slide 1–37

Fetch/Execute — Load



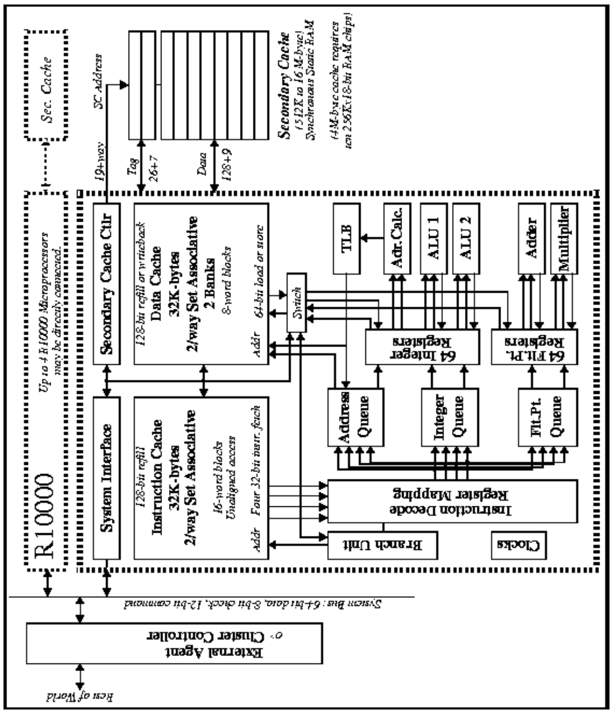
Slide 1–38

Fetch/Execute — Store



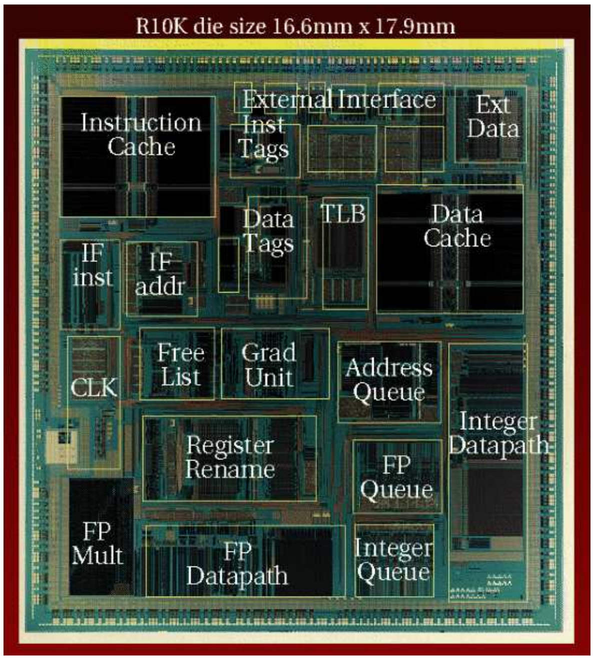
Slide 1–39

MIPS R10000 Block Diagram



Slide 1–40

MIPS R10000 Die Photo



Slide 1–41

Readings and References

- Read Maccabe, sections 3.2, 3.1.

Slide 1–42