What were in CS120B

- General purpose processors (GPP), application specific processors (ASIP), single purpose processor (SPP)
- How to build SPP
  - State machine implementation
- Basics about GPP
  - Fetch, decode, read operands, execute, store results
  - Actions in each cycle
  - Sample instruction set
  - 8051 – 8-bit processor
- Memory hierarch
- Bus architecture, interrupts
- A digital camera example

©1998 Morgan Kaufmann Publishers

What we have in CS161

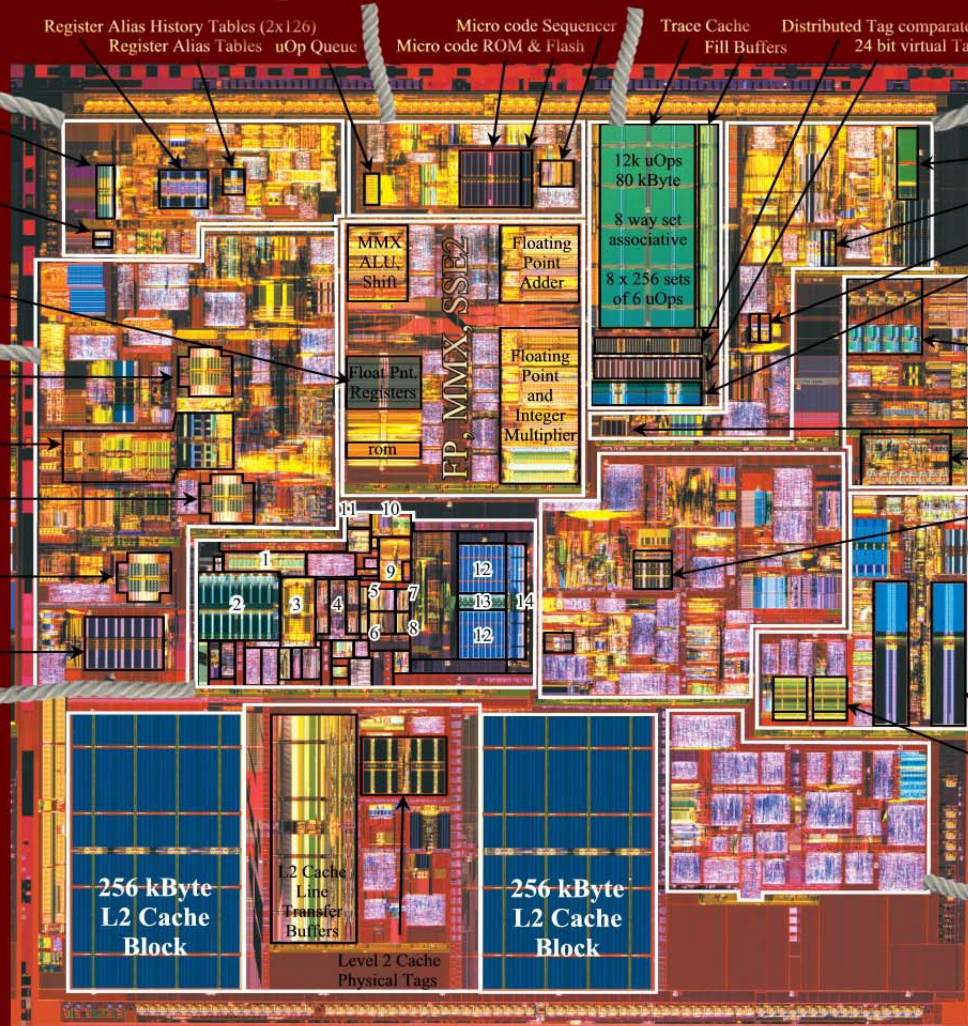
- There will be overlaps. But CS161 goes into more details that are up-to-date.
- A specific instruction set architecture (Chapter 2)
- ~~Arithmetic and how to build an ALU (Chapter 3)~~
- Performance issues (Chapter 4)
- Constructing a processor to execute our instructions, the data path and control (Chapter 5)
- Pipelining to improve performance (Chapter 6)
- Memory: caches and virtual memory (Chapter 7)
- ~~Peripherals (Chapter 8)~~

©1998 Morgan Kaufmann Publishers

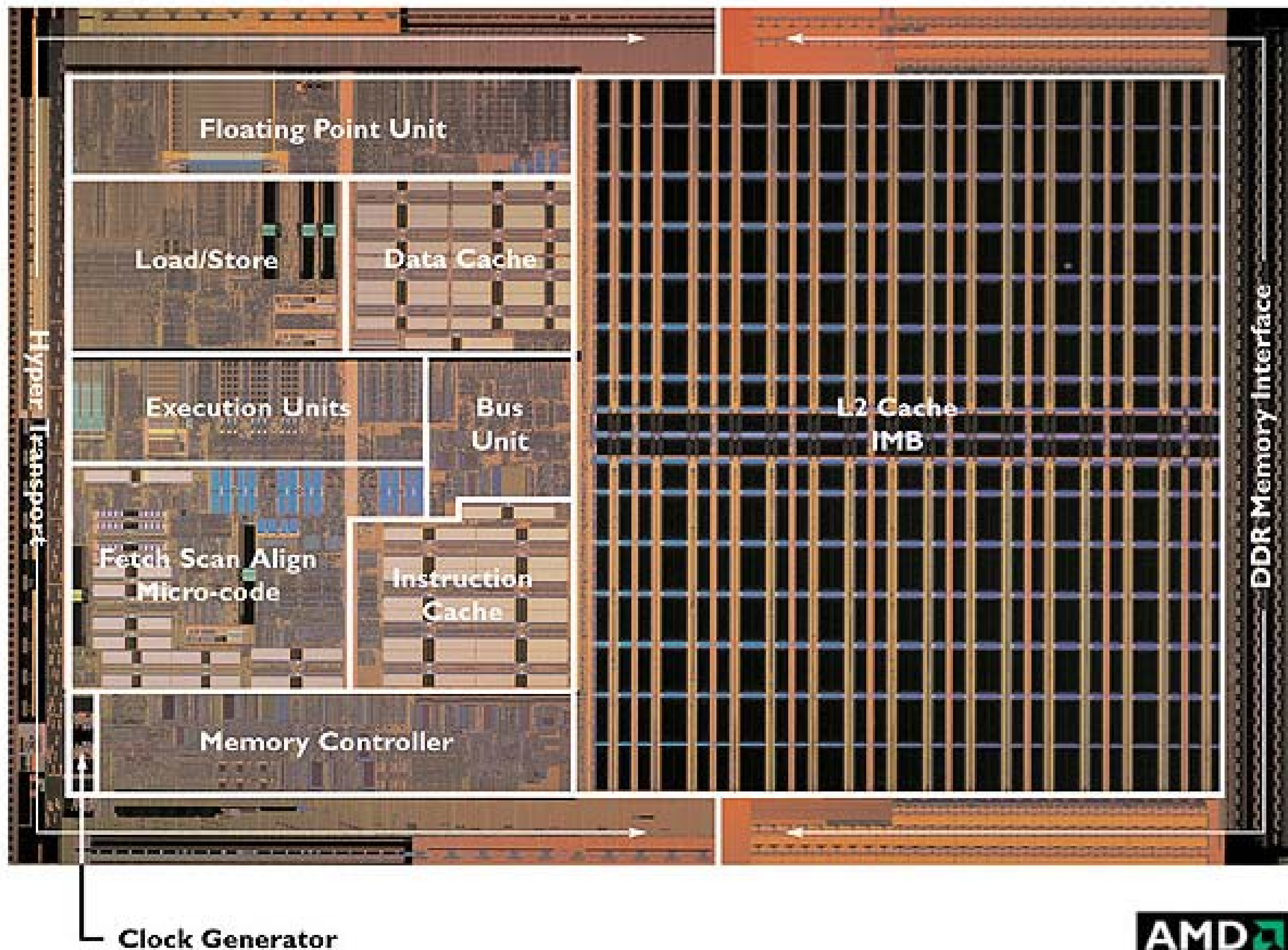
Chapter 1 – Computer Abstractions and Technology

©1998 Morgan Kaufmann Publishers

Introduction



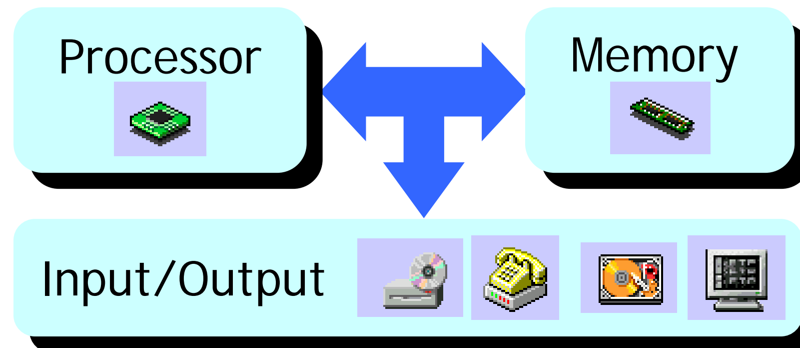
Introduction



UNIVERSITY of CALIFORNIA

What is computer architecture about

- Computer architecture is the study of building computer systems.



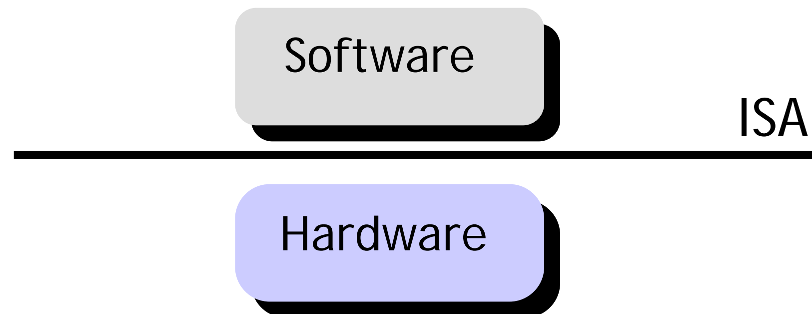
- CS161 is roughly split into three parts.
  - The first third discusses instruction set architectures—the bridge between hardware and software.
  - Next, we introduce more advanced processor implementations. The focus is on pipelining, which is one of the most important ways to improve performance.
  - Finally, we talk about memory systems, I/O, and how to connect it all together.

©1998 Morgan Kaufmann Publishers

Chapter 2 – Instructions: Language of the Computer

©1998 Morgan Kaufmann Publishers

Instruction set architecture



- We′ll talk about several important issues that we didn′t see in the simple processor from CS120B.
  - The instruction set in CS120B lacked many features, such as support for function calls. We′ll work with a larger, more realistic processor.
  - We′ll also see more ways in which the instruction set architecture affects the hardware design.

©1998 Morgan Kaufmann Publishers

MIPS

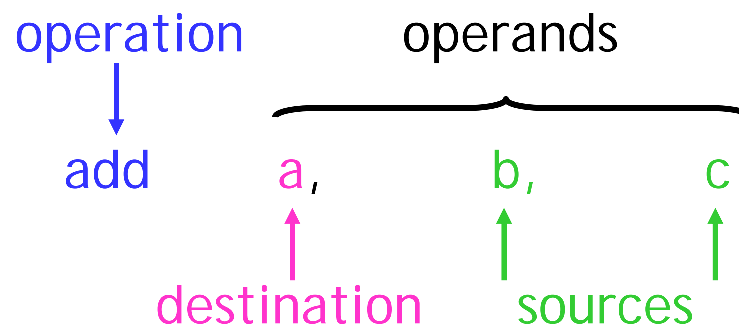
- In this class, we'll use the MIPS instruction set architecture (ISA) to illustrate concepts in assembly language and machine organization
  - Of course, the concepts are not MIPS-specific
  - MIPS is just convenient because it is real, yet simple (unlike x86)
- The MIPS ISA is still used in many places today. Primarily in embedded systems, like:
  - Various routers from Cisco
  - Game machines like the Nintendo 64 and Sony Playstation 2
- You must become "fluent" in MIPS assembly:
  - Translate from C to MIPS and MIPS to C

©1998 Morgan Kaufmann Publishers

MIPS: register-to-register, three address

- **MIPS is a register-to-register, or load/store, architecture.**
  - **The destination and sources must all be registers.**
  - **Special instructions, which we'll see later, are needed to access main memory.**

- **MIPS uses three-address instructions for data manipulation.**
  - **Each ALU instruction contains a destination and two sources.**
  - **For example, an addition instruction (a = b + c) has the form:**



©1998 Morgan Kaufmann Publishers

MIPS register names

- MIPS register names begin with a $. There are two naming conventions:
  - By number:

    $0 $1 $2 … $31

  - By (mostly) two-character names, such as:

    $a0-$a3 $s0-$s7 $t0-$t9 $sp $ra

- Not all of the registers are equivalent:
  - E.g., register $0 or $zero always contains the value 0
    - (go ahead, try to change it)

- Other registers have special uses, by convention:
  - E.g., register $sp is used to hold the "stack pointer"

- You have to be a little careful in picking registers for your programs.

©1998 Morgan Kaufmann Publishers

Policy of Use Conventions

Name	Register number	Usage
$zero	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	Saved temporaries
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	reserved for OS kernel
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

©1998 Morgan Kaufmann Publishers

Basic arithmetic and logic operations

- The basic integer arithmetic operations include the following:

  add sub mul div

- And here are a few logical operations:

  and or xor

- Remember that these all require three register operands; for example:

```
add    $t0, $t1, $t2        # $t0 = $t1 + $t2
xor    $s1, $s1, $a0        # $s1 = $s1 xor $a0
```


©1998 Morgan Kaufmann Publishers

Immediate operands

- The ALU instructions we've seen so far expect register operands. How do you get data into registers in the first place?
  - Some MIPS instructions allow you to specify a signed constant, or "immediate" value, for the second source instead of a register. For example, here is the immediate add instruction, addi:

```
addi    $t0, $t1, 4        # $t0 = $t1 + 4
```

  - Immediate operands can be used in conjunction with the $zero register to write constants into registers:

```
addi    $t0, $0, 4         # $t0 = 4
```

  - Data can also be loaded first into the memory along with the executable file. Then you can use load instructions to put them into registers

```
lw      $t0, 8($t1)        # $t0 = mem[8+$t1]
```

- MIPS is still considered a load/store architecture, because arithmetic operands cannot be from arbitrary memory locations. They must either be registers or constants that are embedded in the instruction.

©1998 Morgan Kaufmann Publishers

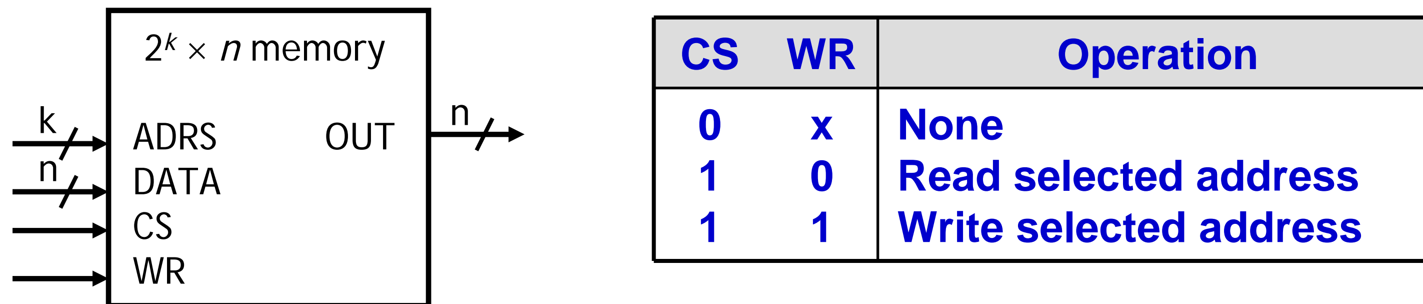
We need more space – memory

- Registers are fast and convenient, but we have only 32 of them, and each one is just 32-bits wide.
  - That's not enough to hold data structures like large arrays.
  - We also can't access data elements that are wider than 32 bits.
- We need to add some main memory to the system!
  - RAM is cheaper and denser than registers, so we can add lots of it.
  - But memory is also significantly slower, so registers should be used whenever possible.
- In the past, using registers wisely was the programmer's job.
  - For example, C has a keyword "register" that marks commonly-used variables which should be kept in the register file if possible.
  - However, modern compilers do a pretty good job of using registers intelligently and minimizing RAM accesses.

©1998 Morgan Kaufmann Publishers

Memory review

- Memory sizes are specified much like register files; here is a 2^k x n RAM.

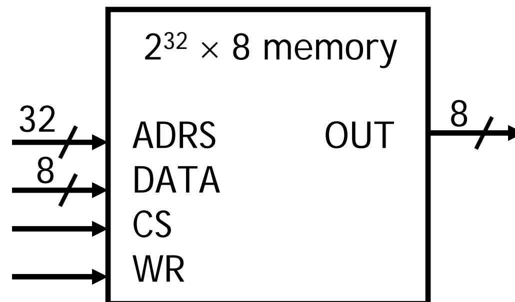


CS	WR	Operation
0	x	None
1	0	Read selected address
1	1	Write selected address

- A chip select input CS enables or “disables” the RAM.
- ADRS specifies the memory location to access.
- WR selects between reading from or writing to the memory.
  - To read from memory, WR should be set to 0. OUT will be the n-bit value stored at ADRS.
  - To write to memory, we set WR = 1. DATA is the n-bit value to store in memory.

©1998 Morgan Kaufmann Publishers

MIPS memory



- MIPS memory is byte-addressable, which means that each memory address references an 8-bit quantity.
- The MIPS architecture can support up to 32 address lines.
  - This results in a 2^{32} x 8 RAM, which would be 4 GB of memory.
  - Not all actual MIPS machines will have this much!

©1998 Morgan Kaufmann Publishers

Bytes and words

- Remember to be careful with memory addresses when accessing words.
- For instance, assume an array of words begins at address 2000.
  - The first array element is at address 2000.
  - The second word is at address *2004*, not 2001.
- For example, if $a0 contains 2000, then

```
lw $t0, 0($a0)
```

  accesses the first word of the array, but

```
lw $t0, 8($a0)
```

  would access the *third* word of the array, at address 2008.

©1998 Morgan Kaufmann Publishers

Loading and storing bytes

- The MIPS instruction set includes dedicated load and store instructions for accessing memory.

- The main difference is that MIPS uses indexed addressing.
  - The address operand specifies a signed constant and a register.
  - These values are added to generate the effective address.
- The MIPS "load byte" instruction lb transfers one byte of data from main memory to a register.

```
lb $t0, 20($a0)        # $t0 = Memory[$a0 + 20]

question: what about the other 3 bytes in $t0?

Sign extension!
```

- The "store byte" instruction sb transfers the lowest byte of data from a register into main memory.

```
sb $t0, 20($a0)        # Memory[$a0 + 20] = $t0
```


©1998 Morgan Kaufmann Publishers

Loading and storing words

- You can also load or store 32-bit quantities—a complete word instead of just a byte—with the lw and sw instructions.

```
lw $t0, 20($a0)        # $t0 = Memory[$a0 + 20]
sw $t0, 20($a0)        # Memory[$a0 + 20] = $t0
```

- Most programming languages support several 32-bit data types.
  - Integers
  - Single-precision floating-point numbers
  - Memory addresses, or pointers
- Unless otherwise stated, we'll assume words are the basic unit of data.

©1998 Morgan Kaufmann Publishers