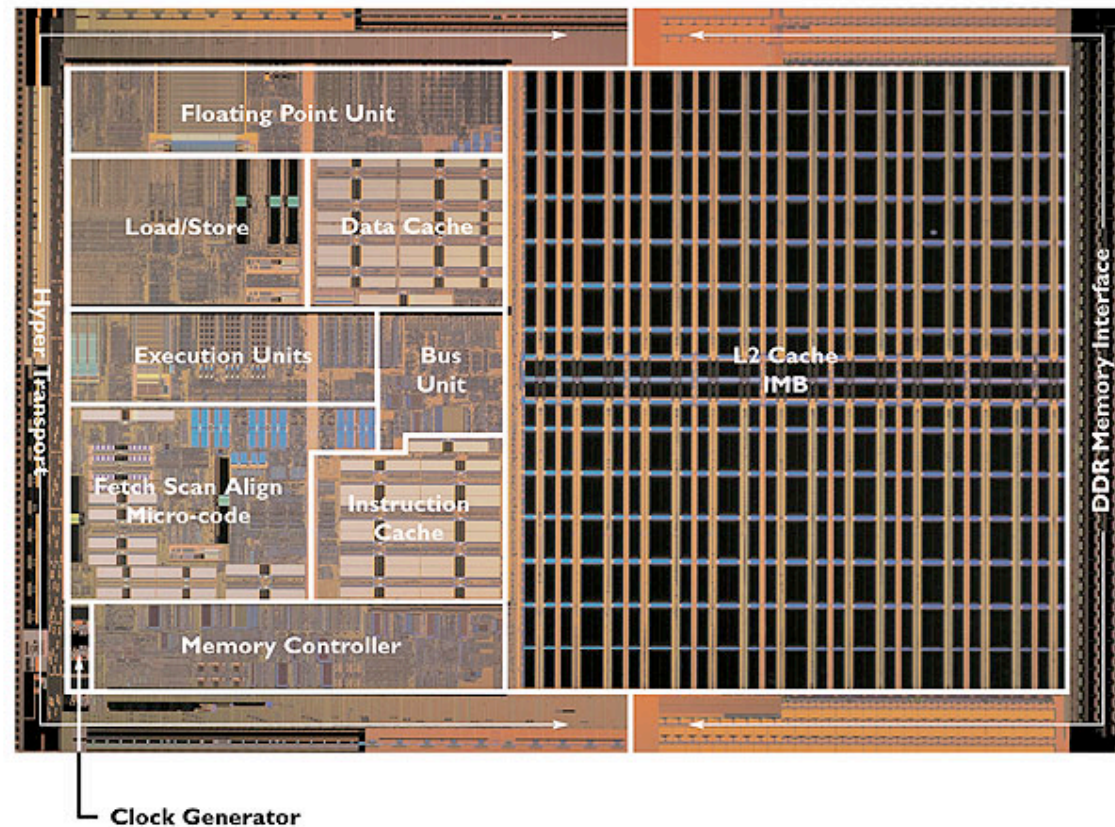


# CS232: Computer Architecture II

Spring 2009



# Instruction set architectures

---



- In CS232, we'll talk about several important issues that we didn't see in the simple processor from CS231.
  - The instruction set in CS231 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.

# 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](#)



# What you will need to learn this month

---

- You must become “fluent” in MIPS assembly:
  - Translate from C to MIPS and MIPS to C
- Example problem from a previous mid-term 1:

Question 3: Write a recursive function (30 points)

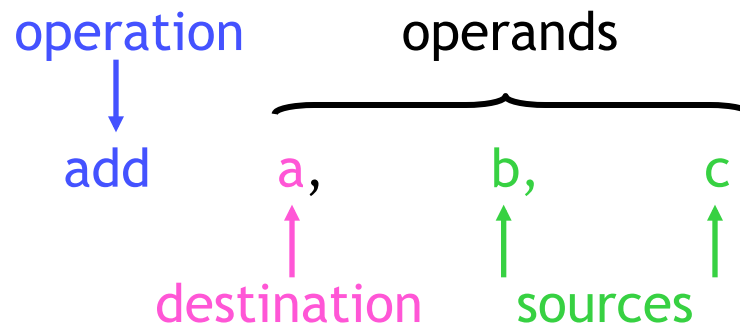
Here is a function pow that takes two arguments (n and m, both 32-bit numbers) and returns  $n^m$  (i.e., n raised to the  $m^{\text{th}}$  power).

```
int
pow(int n, int m) {
    if (m == 1)
        return n;
    return n * pow(n, m-1);
}
```

Translate this into a MIPS assembly language function.

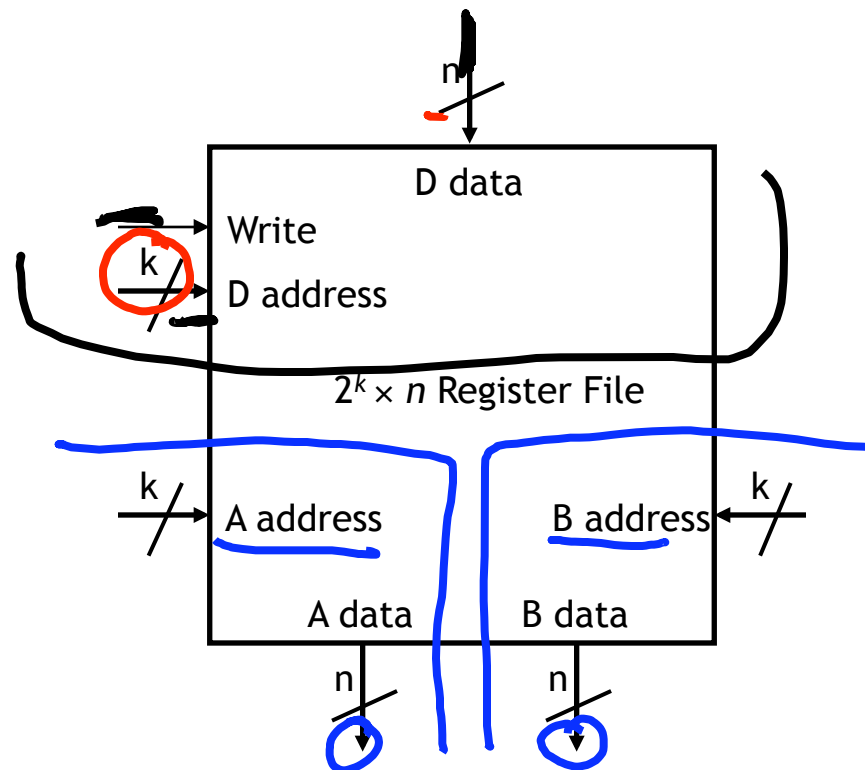
# MIPS: register-to-register, three address

- 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 today, 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:



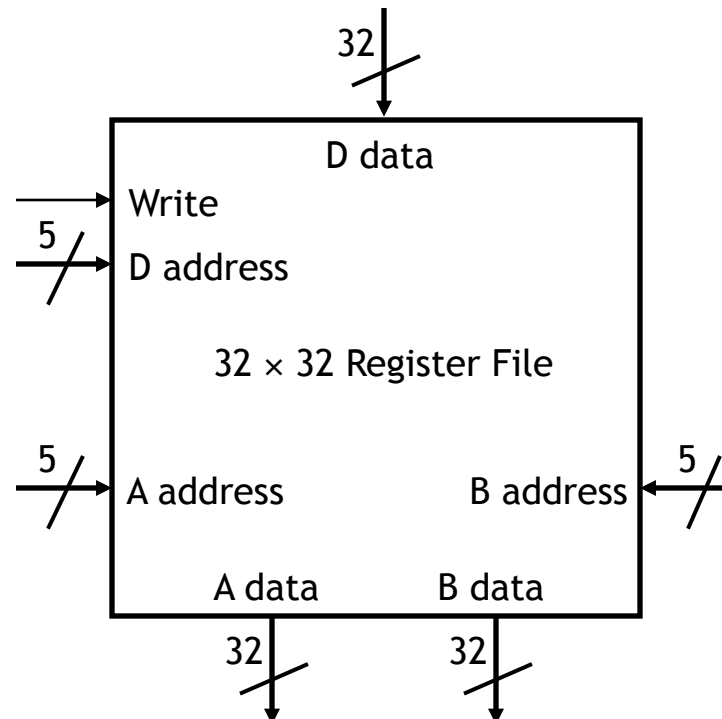
# Register file review

- Here is a block symbol for a general  $2^k \times n$  register file.
  - If **Write = 1**, then **D data** is stored into **D address**.
  - You can read from two registers at once, by supplying the **A address** and **B address** inputs. The outputs appear as **A data** and **B data**.
- Registers are clocked, sequential devices.
  - We can read from the register file at any time.
  - Data is written only on the positive edge of the clock.



# MIPS register file 32b arch

- MIPS processors have 32 registers, each of which holds a 32-bit value.
  - Register addresses are 5 bits long.  $2^5 = 32$
  - The data inputs and outputs are 32-bits wide.
- More registers might seem better, but there is a limit to the goodness.
  - It's more expensive, because of both the registers themselves as well as the decoders and muxes needed to select individual registers.
  - Instruction lengths may be affected, as we'll see in the future.



# 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:

*argument* \$a0-\$a3    \$s0-\$s7    \$t0-\$t9    \$sp    \$ra  
*temporary*

- 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.



# Basic arithmetic and logic operations

- The basic integer arithmetic operations include the following:

add   sub   mul   div

*Use pseudo instructions*

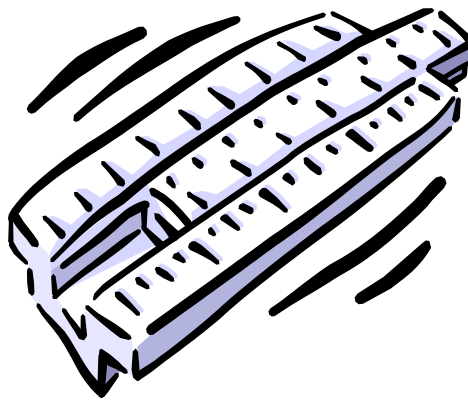
- And here are a few logical operations:

*bit-wise*

and   or   xor

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

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



**Note: a full MIPS ISA reference can be found in Appendix A (linked from website)**

# Larger expressions

- More complex arithmetic expressions may require multiple operations at the instruction set level.

$$\underline{t0} = (\underline{t1 + t2}) \times (\underline{t3 - t4})$$

```
add    $t0, $t1, $t2    # $t0 contains $t1 + $t2
sub    $s0, $t3, $t4    # Temporary value $s0 = $t3 - $t4
mul    $t0, $t0, $s0    # $t0 contains the final product
```

- Temporary registers may be necessary, since each MIPS instructions can access only two source registers and one destination.
  - In this example, we could re-use \$t3 instead of introducing \$s0.
  - But be careful not to modify registers that are needed again later.

# 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  
*add \$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

- 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.

~~*addi \$t0, 1, 2*~~ → *addi \$t0, \$0, 2*

## A more complete example

---

- What if we wanted to compute the following?

$$1 + 2 + 3 + 4$$

addi \$t0, \$0, 1  
addi \$t2, \$t0, 2  
addi \$t1, \$t2, 3  
addi \$t2, \$t2, 4

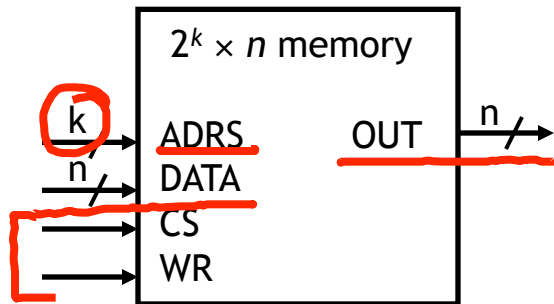
# We need more space!

---

- 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.

# Memory review

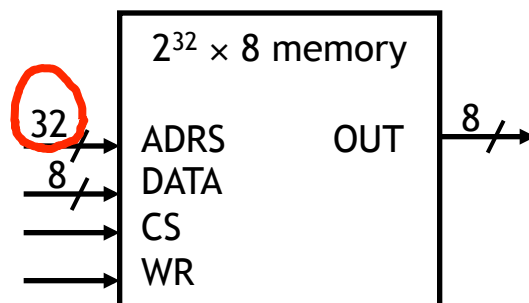
- Memory sizes are specified much like register files; here is a  $2^k \times n$  RAM.  
*read or write*



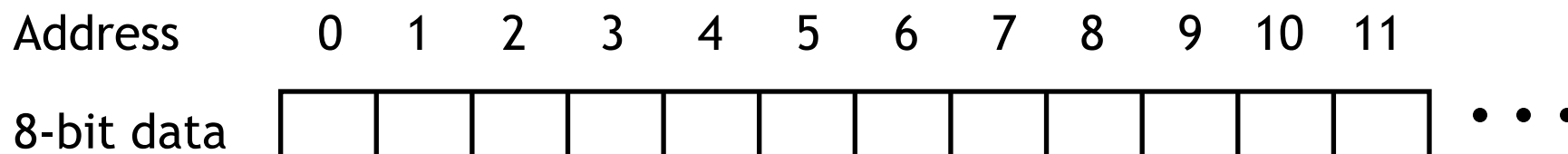
CS	WR	Operation
0	x	None
1	0	Read selected address
1	1	Write selected address

- 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.

# 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!



# Loading and storing bytes

- The MIPS instruction set includes dedicated load and store instructions for accessing memory, much like the CS231 example processor.
- 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]`

- 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`



# Byte loads

---

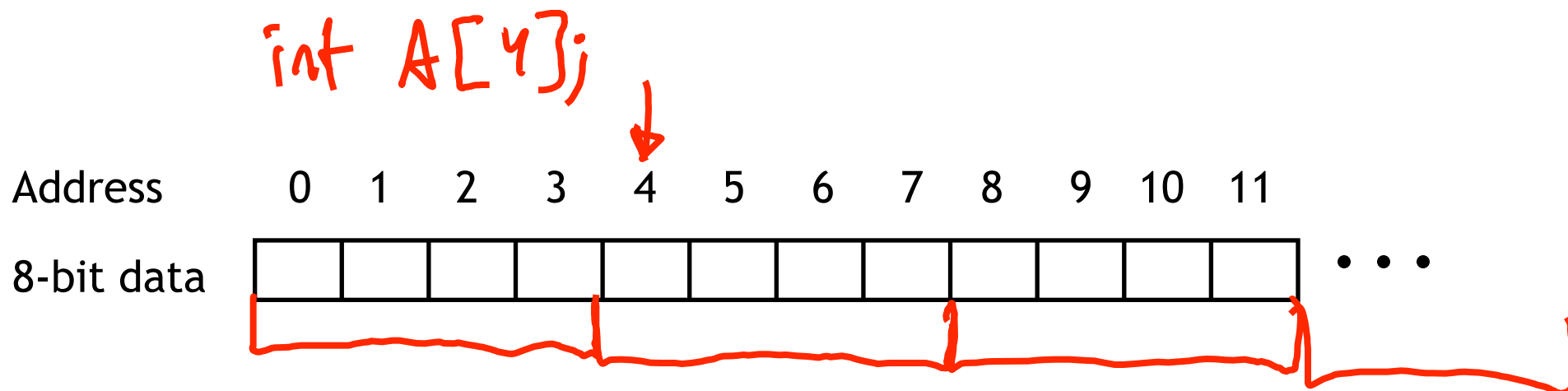
- **Question:** if you load a byte (8 bits) into a register (32 bits), what value do those other 24 bits have?

# 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.



# An array of 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. ←
- Revisiting the earlier 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.

```
lw $t0, 2($a0)
```

# Computing with memory

---

- So, to compute with memory-based data, you must:
  1. Load the data from memory to the register file.
  2. Do the computation, leaving the result in a register.
  3. Store that value back to memory if needed.
- For example, let's say that you wanted to do the same addition, but the values were in memory. How can we do the following using MIPS assembly language?

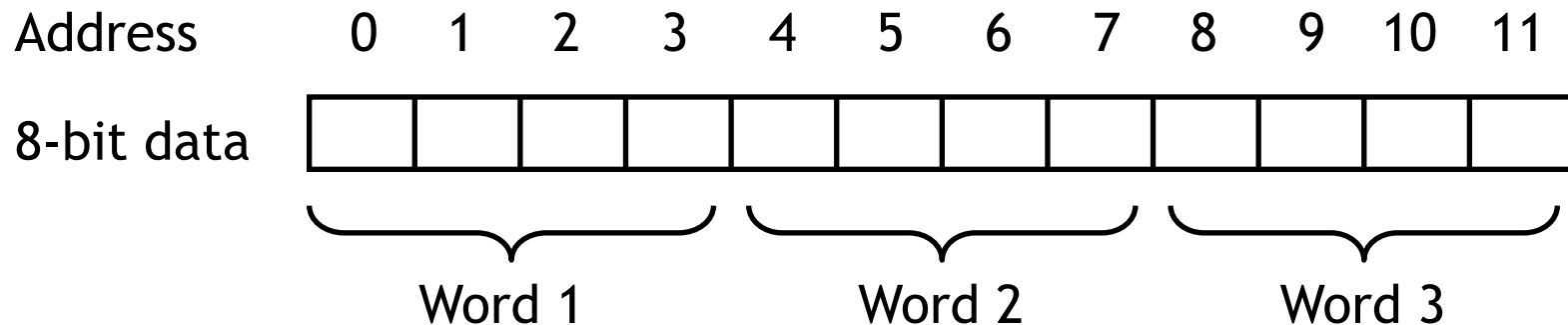
```
char A[4] = {1, 2, 3, 4};
```

```
int result;
```

```
result = A[0] + A[1] + A[2] + A[3];
```

# Memory alignment

- Keep in mind that memory is byte-addressable, so a 32-bit word actually occupies four contiguous locations (bytes) of main memory.



- The MIPS architecture requires words to be **aligned** in memory; 32-bit words must start at an address that is divisible by 4.
  - 0, 4, 8 and 12 are valid **word addresses**.
  - 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
  - Unaligned memory accesses result in a **bus error**, which you may have unfortunately seen before.
- This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

# Next time

---

- Our next topic is **control flow** in MIPS
  - On Monday, we'll introduce **loops**.
  - Next Wednesday, we'll show if/then/else structures.

# General hints to reach CS232 nirvana

---

- **Remember the big picture.**  
What are we trying to accomplish, and why?
- **Read the textbook.**  
It's clear, well-organized, and well-written. The diagrams can be complex, but are worth studying. Work through the examples and try some exercises on your own. Read the “Real Stuff” and “Historical Perspective” sections.
- **Talk to each other.**  
You can learn a lot from other CS232 students, both by asking and answering questions. Find some good partners for the homeworks (but make sure you all understand what's going on).
- **Help us help you.**  
Come to lectures, sections and office hours. Send email or post on the newsgroup. Ask lots of questions! Check out the web page:

<http://www-courses.cs.uiuc.edu/~cs232>