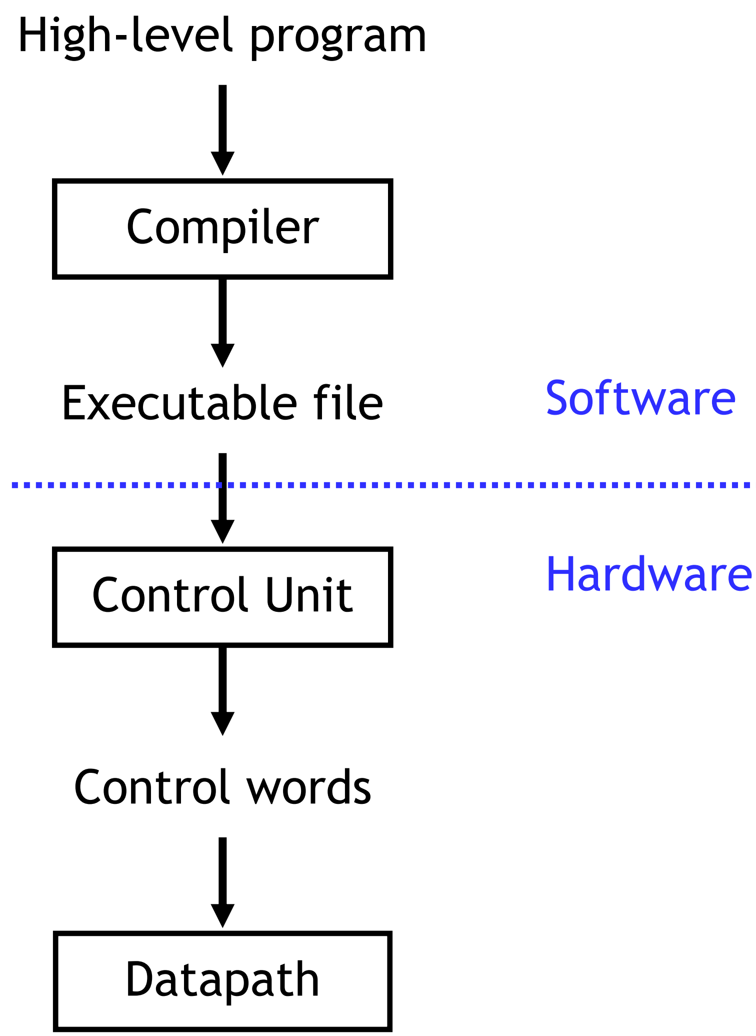
Basic MIPS Architecture

- Today we'll introduce the MIPS processor, which will be our example system for much of this semester.
  - We present the basic instruction set architecture.
  - This also involves some discussion of the CPU hardware.
- This architecture is mostly a superset of the one from CS231, so today's lecture should also serve as a quick review.

Programming and CPUs

- Programs written in a high-level language like C++ must be compiled using tools like CC or gcc.
- The result is an executable program file, containing CPU-specific machine language instructions.
  - These instructions represent functions that can be handled by the processor.
  - When you run the program, the instructions are loaded into memory and executed by the processor.
- Thus, a processor's instruction set is the boundary between software and hardware.



Instruction sets

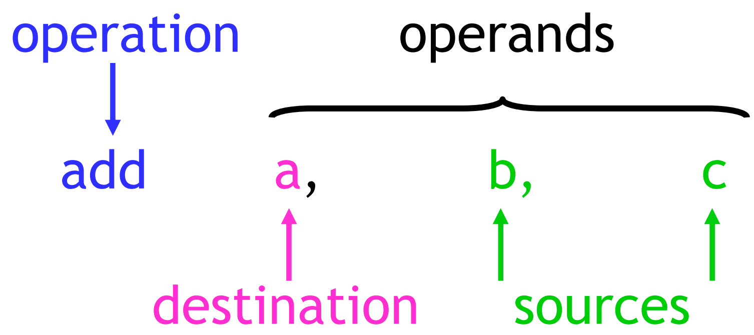
- An instruction set architecture closely reflects the processor's design, so different CPUs have different instruction sets.
- Older processors used complex instruction sets, or CISC architectures.
  - Many powerful instructions were supported, making the assembly language programmer's job much easier.
  - But this meant that the processor was more complex, which made the hardware designer's life a bloody nightmare.
- Many new processors use reduced instruction sets, or RISC architectures.
  - Only relatively simple instructions are available. But with high-level languages and compilers, the impact on programmers is minimal.
  - On the other hand, the hardware is much easier to design, optimize, and teach in classes.
- Even most current CISC processors, such as Intel 8086-based chips, are now implemented using a lot of RISC techniques.

MIPS

- MIPS was one of the first RISC architectures. It was started about 20 years ago by John Hennessy, one of the authors of our textbook.
- The architecture is similar to that of other recent CPU designs, including Sun's SPARC, IBM and Motorola's PowerPC, and ARM-based processors.
- MIPS designs are still used in many places today.
  - Silicon Graphics workstations and servers
  - Various routers from Cisco
  - Game machines like the Nintendo 64 and Sony Playstation 2.

MIPS: three address, register-to-register

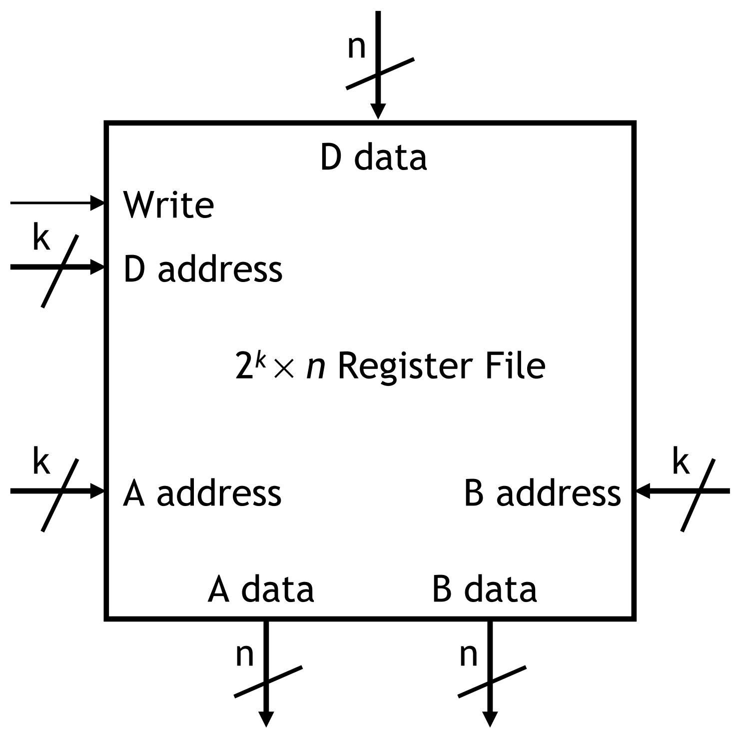
- MIPS uses three-address instructions for data manipulation.
  - Each ALU instruction contains a destination and two sources.
  - For example, an addition instruction (a = b + c) has the form:



- 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.

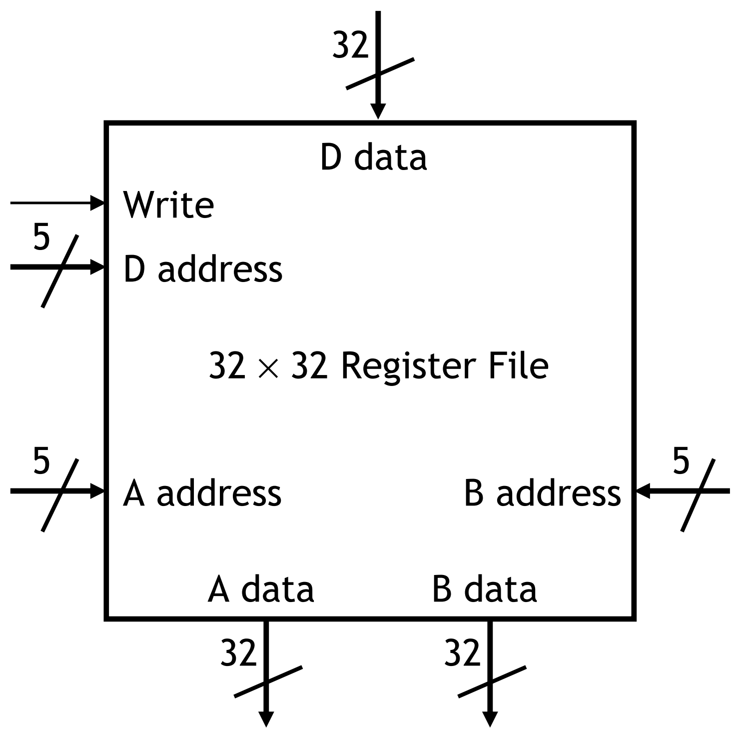
Register file review

- 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

- MIPS processors have 32 registers, each of which holds a 32-bit value.
  - Register addresses are 5 bits long.
  - 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 on Wednesday.



MIPS register names

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

    $0 $1 $2 … $31

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

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

- Not all of these are general purpose registers.
  - Some have specific uses that we'll see later.
  - You have to be careful in picking registers for your programs.

Basic arithmetic and logic operations

- The basic integer arithmetic operations include the following:

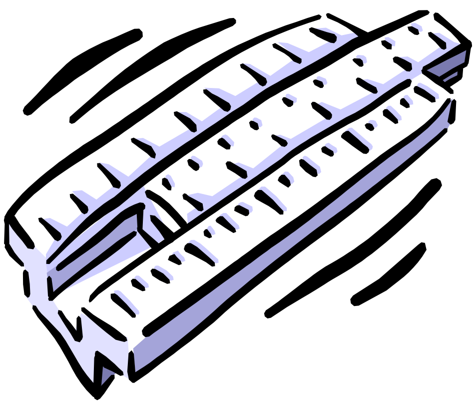
  add sub mul div

- 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
mul  $s1, $s1, $a0      # $s1 = $s1 × $a0
```



Larger expressions

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

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

```
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
```

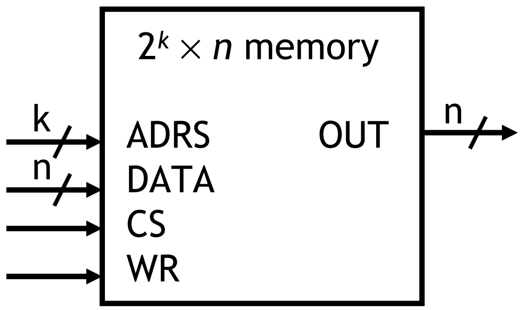
- Temporary registers may be necessary, since each MIPS instructions can access only two source registers and one destination.
  - In this example we could re-use $t3 instead of introducing $s0.
  - But be careful not to modify registers that are needed again later.

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

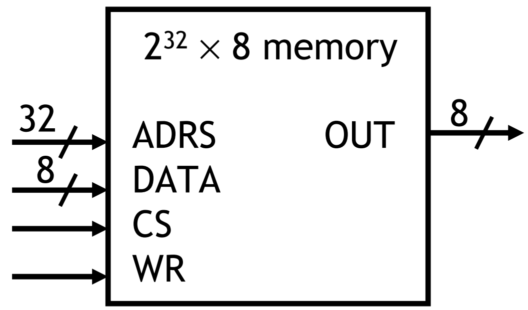
- Memory sizes are specified much like register files; here is a $2^k \times n$ RAM.



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
```

Indexed addressing and arrays

```
lb $t0, const($a0)
```

- Indexed addressing is good for accessing contiguous locations of memory, like arrays or structures.
  - The constant is the base address of the array or structure.
  - The register indicates the element to access.
- For example, if $a0 contains 0, then

```
lb $t0, 2000($a0)
```

  reads the first byte of an array starting at address 2000.
- If $a0 contains 8, then the same instruction would access the ninth byte of the array, at address 2008.
- This is why array indices in C and Java start at 0 and not 1.

Arrays and indexed addressing

```
lb $t0, const($a0)
```

- You can also reverse the roles of the constant and register. This can be useful if you know exactly which array or structure elements you need.
  - The register could contain the address of the data structure.
  - The constant would then be the index of the desired element.
- For example, if $a0 contains 2000, then

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

  accesses the first byte of an array starting at address 2000.
- Changing the constant to 8 would reference the ninth byte of the array, at address 2008.

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

Loading and storing words

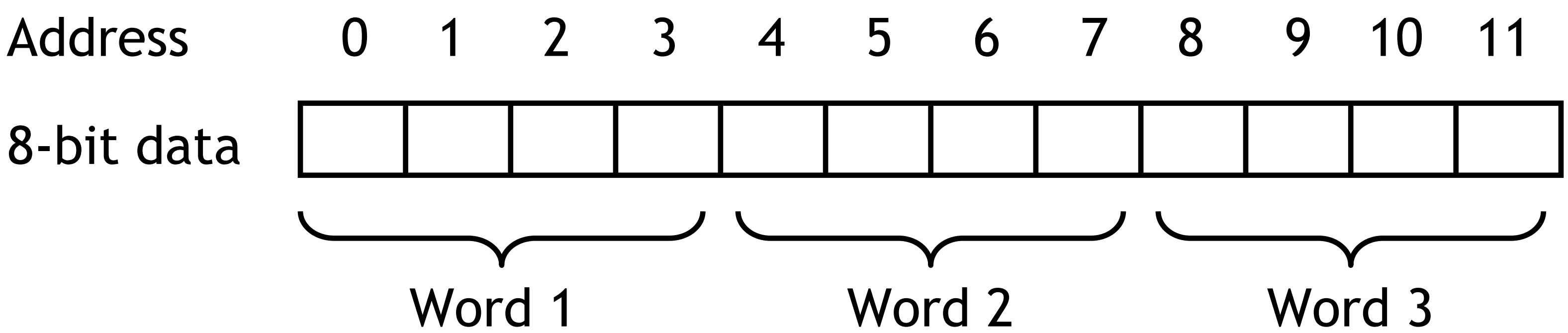
- You can also load or store 32-bit quantities—a complete word instead of just a byte—with the lw and sw instructions.

```
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.

Memory alignment

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

The array example revisited

- 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.

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 an integer array A starts at address 4096. How can we do the following using MIPS assembly language?

```
A[2] = A[1] × A[1]
```

- The solution below assumes that register $t0 contains 4096. (Next week we'll talk about how to get 4096 into $t0 in the first place.)

```
lw   $s0, 4($t0)        # $s0 = A[1]
mul  $s0, $s0, $s0      # $s0 = A[1] × A[1]
sw   $s0, 8($t0)        # A[2] = A[1] × A[1]
```

Summary

- Instruction sets serve as the link between programs and processors.
  - High-level programs must be translated into machine code.
  - Each machine instruction is then executed by the processor.
- We introduced the MIPS architecture.
  - The MIPS processor has thirty-two 32-bit registers.
  - Three-address, register-to-register instructions are used.
  - Loads and stores use indexed addressing to access RAM.
  - Memory is byte-addressable, and words must be aligned.
- Next time we'll discuss control flow and some new instructions that will let us write more interesting programs.