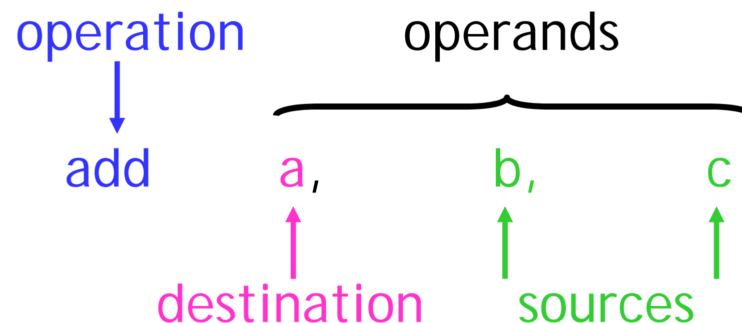
MIPS: register-to-register, three address

- MIPS is a register-to-register, or load/store, architecture
  - destination and sources of instructions must all be registers
  - special instructions to access main memory (later)

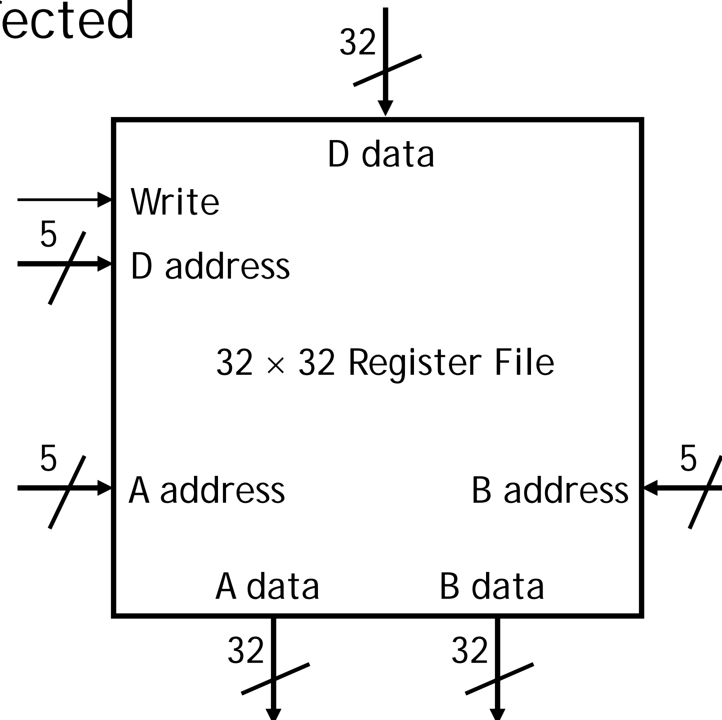
- MIPS uses three-address instructions for data manipulation
  - each ALU instruction contains a destination and two sources

- For example, an addition instruction (a = b + c) has the form:



MIPS register file

- MIPS processors have 32 registers, each of which holds a 32-bit value
  - register addresses are 5 bits long
- More registers might seem better, but there is a limit to the goodness:
  - more expensive: because of registers themselves, plus extra hardware like muxes to select individual registers
  - instruction lengths may be affected



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
  - some have special uses, by convention (**$sp** holds "stack pointer")

- You have to be a little careful in picking registers for your programs
  - for now, stick to the registers **$t0-$t9**

Basic arithmetic and logic operations

- The basic integer arithmetic operations include the following:

add sub mul div

- And here are a few bitwise operations:

and or xor nor

- 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
```

Larger expressions

- Complex arithmetic expressions may require multiple MIPS operations
- Example: **t0 = (t1 + t2) × (t3 – t4)**

```
add  $t0, $t1, $t2    # $t0 contains $t1 + $t2
sub  $t6, $t3, $t4    # temp value $t6 = $t3 - $t4
mul  $t0, $t0, $t6    # $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 **`$t3`** instead of introducing **`$t6`**
  - must be careful not to modify registers that are needed again later

How are registers initialized?

- Special MIPS instructions allow you to specify a signed constant, or "immediate" value, for the second source instead of a register
  - e.g., 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
```

Shorthand: `li    $t0, 4          # $t0 = 4`

(pseudo-instruction)

- 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.

Our first MIPS program

- Let's translate the following C++ program into MIPS:

```
void main() {
  int i = 516;
  int j = i*(i+1)/2;
  i = i + j;
}
```

```
      main:                          # start of main
        li     $t0, 516              # i = 516
        addi   $t1, $t0, 1           # i + 1
        mul    $t1, $t0, $t1         # i * (i + 1)
        li     $t2, 2
        div    $t1, $t1, $t2         # j = i*(i+1)/2
        add    $t0, $t0, $t1         # i = i + j

        jr     $ra                   # return
```

Translate this program into MIPS

- Program to swap two numbers *without* using a temporary:

```
void main() {
  int i = 516;
  int j = 615;
  i = i ^ j;   // ^ = bitwise XOR
  j = i ^ j;
  i = i ^ j;
}
```

Instructions

- MIPS assemblers support pseudo-instructions
  - give the illusion of a more expressive instruction set
  - actually translated into one or more simpler, “real” instructions
- Examples li (load immediate) and move (copy one register into another)
- You can always use pseudo-instructions on assignments and on exams
- For now, we’ll focus on real instructions… how are they encoded?
  - Answer: As a sequence of bits (machine language)
- The control unit sits inside an endless loop:

MIPS instructions

- MIPS machine language is designed to be easy to fetch and decode:
  - — each MIPS instruction is the same length, 32 bits
  - — only three different instruction formats, with many similarities
  - — format determined by its first 6 bits: operation code, or *opcode*

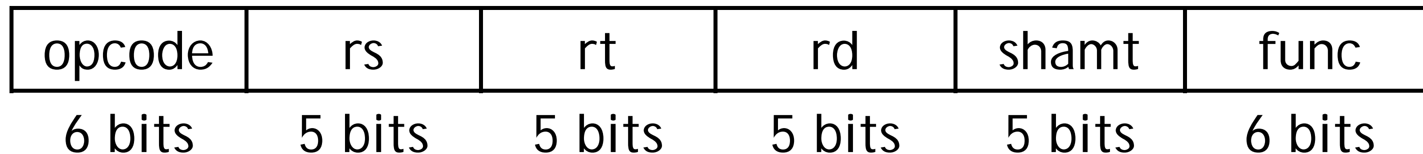
- Fixed-size instructions:
  - (+) easy to fetch/pre-fetch instructions
  - (–) limits number of operations, limits flexibility of ISA

- Small number of formats:
  - (+) easy to decode instructions (simple, fast hardware)
  - (–) limits flexibility of ISA

- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

R-type format

- Register-to-register arithmetic instructions use the R-type format



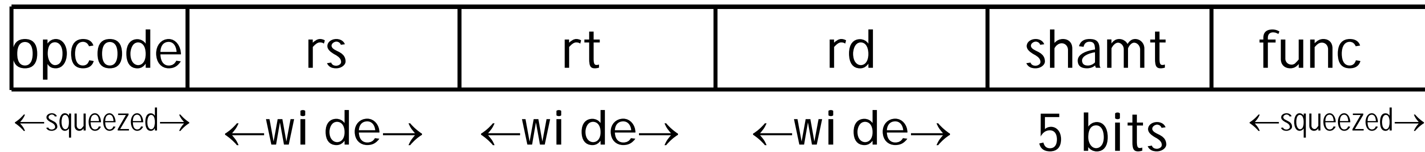
opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Six different fields:
  - opcode is an operation code that selects a specific operation
  - rs and rt are the first and second source registers
  - rd is the destination register
  - shamt is only used for shift instructions
  - func is used together with opcode to select an arithmetic instruction

- The inside back cover of the textbook lists opcodes and function codes for all of the MIPS instructions

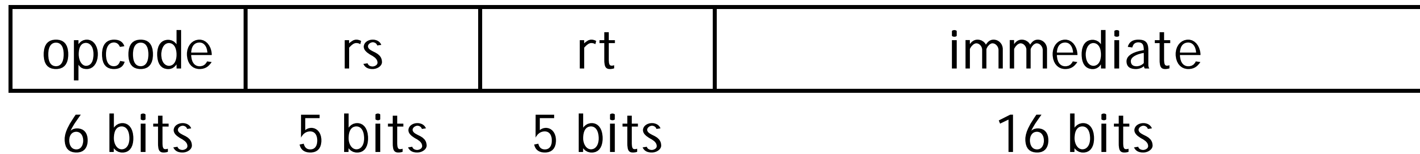
MIPS registers

- We have to encode register names as 5-bit numbers from 00000 to 11111
  – e.g., $t8 is register $24, which is represented as 11000
- The number of registers available affects the instruction length:
  – R-type instructions references 3 registers: total of 15 bits
  – Adding more registers either makes instructions longer than 32 bits, or shortens fields like opcode (reducing number of available operations)



I-type format

- Used for immediate instructions, plus load, store and branch



opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- For uniformity, opcode, rs and rt are located as in the R-format

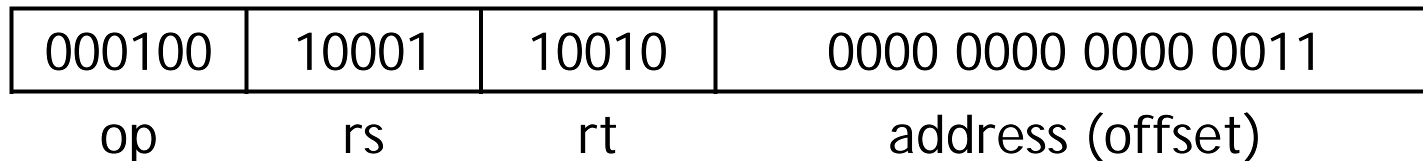
- The meaning of the register fields depends on the exact instruction:
  - rs is *always* a source register (memory address for load and store)
  - rt is a *source* register for store and branch, but a *destination* register for all other I-type instructions

- The immediate is a 16-bit signed two's-complement value.
  - It can range from -32,768 to +32,767.
  - Question: How does MIPS load a 32-bit constant into a register?
  - Answer: Two instructions. **Make the common case fast.**

Branches

- Two branch instructions:

```
beq  $t0, $t1, label   # if t0 == t1, jump to "label"
bne  $t0, $t1, label   # if t0 != t1, jump to "label"
```



000100	10001	10010	0000 0000 0000 0011
op	rs	rt	address (offset)

- For branch instructions, the constant field is not an address, but an *offset* from the current program counter (PC) to the target address.

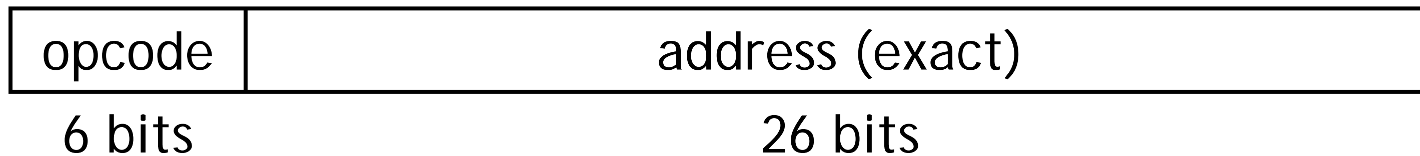
```
      beq   $t0, $t1, EQ
      add   $t0, $t0, $t1
      addi  $t1, $t0, $0
EQ:   add   $v1, $v0, $v0
```

- Since the branch target EQ is *three* instructions past the beq, the address field contains 3

J-type format

- In real programs, branch targets are less than 32,767 instructions away
  - branches are mostly used in loops and conditionals
  - programmers are taught to make loop bodies short

- For "far" jumps, use **j** and **jal** instructions (J-type instruction format)



  - address is always a multiple of 4 (32 bits per instruction)
  - only the top 26 bits actually stored (last two are always 0)

- For even longer jumps, the jump register (jr) instruction can be used.

```
jr $ra        # Jump to 32-bit address in register $ra
```

Pseudo-branches

- The MIPS processor only supports two branch instructions, beq and bne, but to simplify your life the assembler provides the following other branches:

```
blt     $t0, $t1, L1 // Branch if $t0 < $t1
ble     $t0, $t1, L2 // Branch if $t0 <= $t1
bgt     $t0, $t1, L3 // Branch if $t0 > $t1
bge     $t0, $t1, L4 // Branch if $t0 >= $t1
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.

- Later this semester we'll see how supporting just beq and bne simplifies the processor design.