Computer Science 220
Assembly Language & Comp. Architecture
Siena College
Fall 2011

Topic Notes: MIPS Instruction Set Architecture

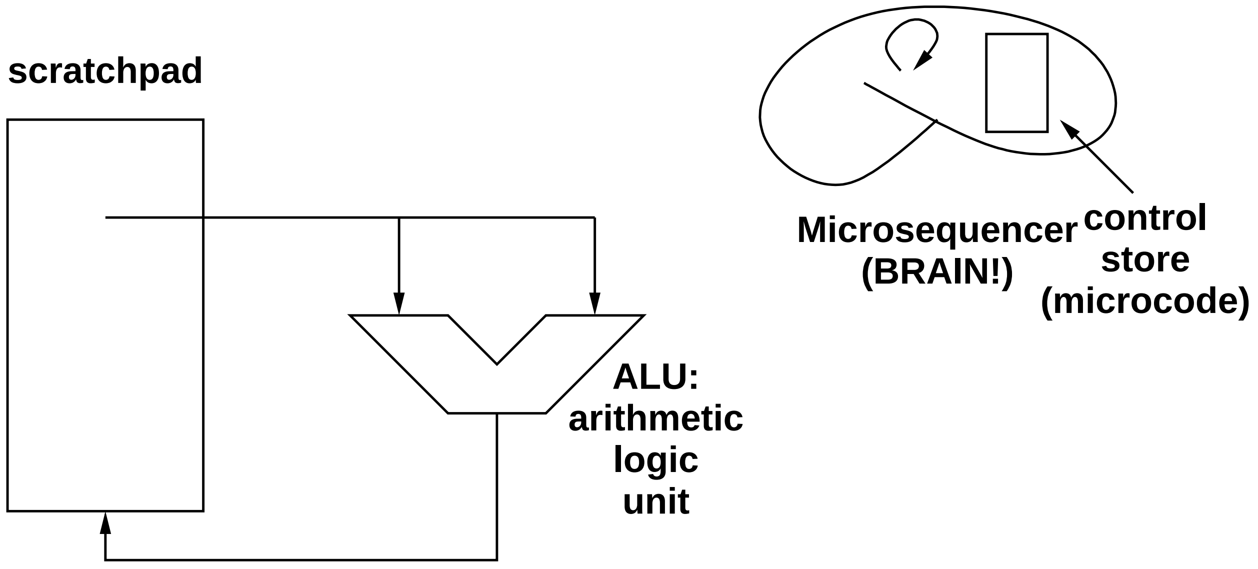
vonNeumann Architecture

Modern computers use the *vonNeumann architecture*.

Idea: a set of instructions and a loop that executes those instructions:

1. Fetch an instruction
2. Update next instruction location
3. Decode the instruction
4. Execute the instruction
5. GOTO 1

Basic picture of the system:



The ALU knows how to do some set of arithmetic and logical operations on values in the scratchpad.

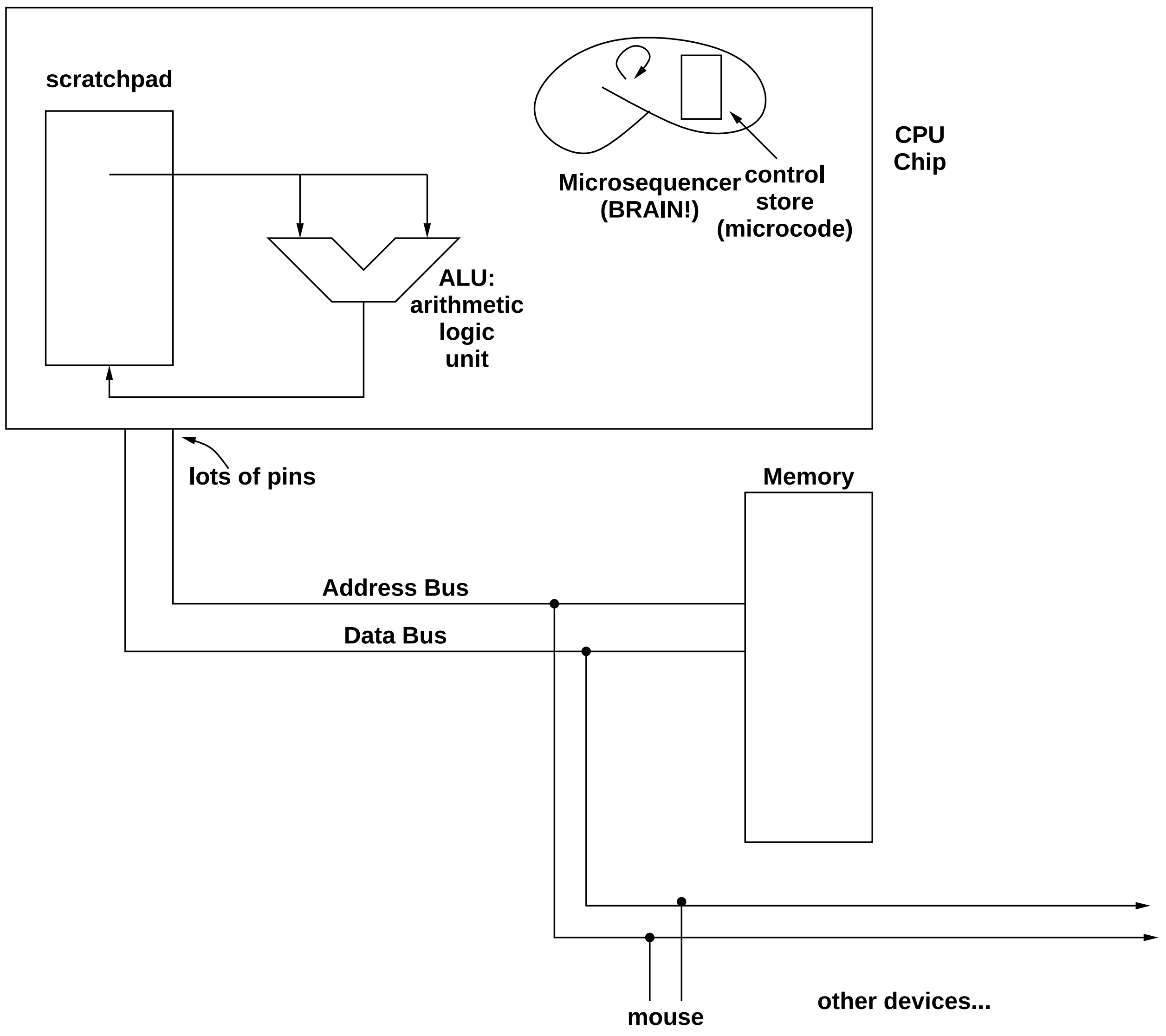
Usually the scratchpad is made up of a set of *registers*.

The micro-sequencer “brain” controls what the ALU reads from the scratchpad and where it might put results, and when.

We will get into details of the micro-sequencer later.

This is what makes up the *central processing unit (CPU)*.

Expand this idea a bit:



CPU interacts with memory and other devices on *buses*.

These buses just carry signals that represent the data. More on these later, too.

We'll have to worry about how we can connect the CPU, memory, other devices to these buses.

There are a variety of speeds, startup rates:

- mouse, keyboard: slow
- disk, network: fast

Some of these are the concern of hardware, others more for the operating system.

MIPS Instruction Set Architecture

We will look at the MIPS *instruction set architecture (ISA)*.

Recall from our first day that the ISA is a key abstraction:

- interface between hardware and low-level software
- standardizes instructions (the languages of the machine), machine language bit patterns, etc.

This idea has many advantages, one being that we can have different underlying hardware implementations of the same ISA. Some may be more efficient (faster) than others, some may be slower but cheaper or easier to construct.

On the down side, sticking to an ISA can sometimes slow innovation. A manufacturer may not wish to change the ISA, especially for a successful ISA that already has lots of software.

Discussion: how important is binary compatibility?

The definition of the ISA is like a contract: it lays out a set of instructions. Software can use those instructions and expect that any hardware implemenation of the ISA will execute those instructions correctly.

This is very similar to the way you can use a class someone else has written in a language like Java (think: `ArrayList`). You can trust that it does what it says it does and does it correctly, but it doesn't matter how it works internally.

MIPS processors are in extensive use by NEC, Nintendo, Cisco, SGI, Sony, etc.

MIPS is an example of *reduced instruction set computer (RISC) architecture*.

RISC architectures have a fewer number of simple instructions than *complex instruction set computer (CISC)* architectures.

Later, we will discuss the relative advantages of the RISC and CISC approaches.

For now:

- Good news: not many instructions or addressing modes to learn
- Bad news: a single instruction performs only a very simple operation, so programming a task takes more instructions
- More good news: all modern ISAs have similar types of instructions, so what we learn for MIPS will let you quickly learn more about any ISA

MIPS Arithmetic Instructions

MIPS arithmetic instructions have three operands:

```
add a, b, c
```

This instruction takes the sum of scratchpad values `b` and `c` and puts the answer into scratchpad value `a`.

It is equivalant to the C code:

```
a = b + c;
```

What if we want to code the following:

```
a = b + c + d;
```

We need to do it in two steps:

```
add a, b, c
add a, a, d
```

Note that multiple operands may refer to the same scratchpad location.

The `sub` instruction is similar.

MIPS Registers and Memory

In MIPS, the operands must be registers. It is this collection of registers (the *register file*) that forms the scratchpad.

- 32 registers are provided
- each register stores a 32-bit value
- compilers associate program variables with registers
- registers are referred to by names such as `$s0` and `$t1`
- we use the "`s`" registers for values that correspond to variables in our programs
- we use the "`t`" registers for temporary values (more on this later)

For example, consider this example from the text:

```
f = (g + h) - (i + j);
```

We choose (or better yet, a compiler chooses) registers to store the values of our variables: `f` in `$s0`, `g` in `$s1`, `h` in `$s2`, `i` in `$s3`, and `j` in `$s4`.

We'll also need two temporary values, which we will store in `$t0` and `$t1`.

The MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

What if you need more variables than there are registers?

Access *memory*!

- think of memory as a large, one-dimensional array
- a memory *address* is an index into this array of values
- memory is a much larger storage space than registers, but access to that space is slower
- MIPS arithmetic (and other) instructions can't operate directly on values in memory
- data must be transferred first from memory into a register, then the answer transferred back

Since registers are 32-bit (4-byte) values, we often access memory in words instead of bytes.

- 2^{32} bytes with byte addresses from 0 to $2^{32} - 1$
- 2^{30} words with byte addresses $0, 4, 8, ...2^{32} - 4$
- words must be aligned on 4-byte boundaries

So suppose we have the following C code to translate to MIPS:

```
A[12] = h + A[8];
```

where `A` is an array of word-sized values.

We have the address of the first element of `A` in register `$s3` and `h` has been assigned to `$s2`.

First, note that the values in the array `A` are word-sized, so each entry takes 4 bytes. We can find entries in the array:

```
 A[0]   $s3+0
 A[1]   $s3+4
 A[2]   $s3+8
  …       …
 A[8]  $s3+32
  …       …
A[12]  $s3+48
```

The notation to get the value at location `$s3+4`, for example, is `4($s3)`.

So what we'd like to write:

```
add 48($s3), $s2, 32($s3)
```

But we can't, since MIPS arithmetic instructions can't operate on values in memory. We'll have to copy the value `A[8]` into a temporary register, add into a temporary register, then store the value in `A[12]`.

The code:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48($s3)
```

The new instructions are to load a word `lw` and store a word `sw`.

Aside: why is it OK to overwrite the value in `$t0` in the `add` instruction even though our original C code doesn't change `A[8]`?

The address in `$s3` is called a *base register* and the constants we add to it are called the *offsets*.

Immediate Addressing Mode

We often need to deal with constants. So far, the only way we'd be able to add a constant to a register is by having that constant in a register or a memory location (and how exactly would we get it there?).

So there is a special format of the `add` instruction: add immediate, specified as `addi`. Its third operand is a constant value used in the addition.

```
addi $s2, $s2, 4
```

MIPS Machine Language

MIPS assembly instructions correspond to 32-bit MIPS machine instructions.

For example:

```
add $t1, $s1, $s2
```

This corresponds to the machine instruction

```
00000010001100100100000000100000
```

Somehow, the fact that this is an `add` instruction and which registers are involved is encoded in this particular 32-bit value.

We interpret the 32-bit value in this case by breaking it down into *fields* according to the *instruction format*.

`op`	`rs`	`rt`	`rd`	`shmat`	`funct`
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
`0`	`17`	`18`	`8`	`0`	`32`
`000000`	`10001`	`10010`	`01000`	`00000`	`100000`

The meanings of the fields:

- `op`: the *opcode* – 6 bits
- `rs`: the first register source operand – 5 bits (why?)
- `rt`: the second register source operand – 5 bits
- `rd`: the register destination operand – 5 bits
- `shmat`: the shift amount — 5 bits (more later)
- `funct`: the variant of the operation – 6 bits

This is an example of an *R-type* (register) instruction, which is encoded in the *R-format*. These are the instructions that require three registers to be specified.

The 32 registers are encoded as follows:

Name	**Register Number**	**Usage**
`$zero`	0	constant value 0
`$at`	1	reserved for assembler use
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	procedure parameters
`$t0-$t7`	8-15	temporary variables
`$s0-$s7`	16-23	saved variable values
`$t8-$t9`	24-25	more temporary variables
	26-27	reserved for operating system use
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Other instructions don't need three registers. Immediate mode instructions, for example, need 2 registers and a constant value. These are *I-type* instructions, stored in the *I-format*:

```
addi $s1, $s2, 100
```

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits
8	17	18	100
001000	10001	10010	0000000000110100

Here, three of the fields are replaced by a single 16-bit field called `address`. For the `addi` instruction, this stores the constant value to be added.

The load and store instructions use this format as well.

```
lw $t0, 1200($t1)
```

This instruction's function is to retrieve the value from memory at the address pointed to by the contents of `$t1`, offset by 1200, and store the value in `$t0`.

op	rs	rt	address
35	9	8	1200
010101	01001	01000	0000010010110000

The `sw` instruction is similar, with opcode 43.

MIPS Logical Instructions

We will look quickly at the logical shift instructions: `sll` and `srl`, which stand for shift left logical and shift right logical, respectively.

These instructions use the `shamt` field in an R-format instruction:

```
sll $t2, $s0, 4
```

op	rs	rt	rd	shmat	funct
0	0	16	10	4	0

Note that `rs` is not used.

Recall that these are quick ways to multiply and divide by powers of 2.

Bitwise `and`, `or`, `nor` follow the R-format, much like `add`, and the immediate versions `andi` and `ori` follow the I-format, like `addi`.

MIPS Control Flow Instructions

Any non-trivial program needs to make decisions, hence the *conditional branch* instructions:

```
beq reg1, reg2, label
bne reg1, reg2, label
```

`beq` will cause a branch to the statement labeled `label` if the values of `reg1` and `reg2` are equal, and continue to the next instruction otherwise.

`bne` branches when not equal.

These use the I-format for the machine instruction:

```
bne $s0, $s1, Exit
```

op	rs	rt	address
5	16	17	Exit

The address has to fit in 16 bits, so does this mean we can only branch to a 16-bit address? No - we usually use the `address` field as a relative offset to the program counter (`PC`): *PC-relative addressing*.

So if the label `Exit` is 44 away in the positive direction from the current program counter, we store 11 in the address field.

We divide by 4 since we know the bottom 2 bits are 0's anyway (addresses are word-addressable). This means we can jump anywhere from -2^{17} to 2^{17} from the current `PC`.

Note: the `PC` is incremented (by 4, as instructions are word-sized) early in the execution of a given instruction. (Step 2 of Fetch, Update, Decode, Execute). Therefore, by the time we're really executing an instruction, it contains the address of the *next* instruction to be executed. A branch that is taken simply needs to modify the `PC` before we fetch the next instruction.

There is also an unconditional jump instruction:

```
j label
```

No registers here, so we have more bits available for the address. This is a *J-format* instruction.

If we want to jump to memory location 4848, the instruction is:

op	address
6 bits	26 bits
2	1212

Again, the bottom 2 bits are always 0, so we divide our intended jump target by 4 when encoding the instruction.

We can also perform inequality comparisons with two more instructions:

```
slt $t1, $s2, $s1
slti $t2, $t4, 8
```

These are set on less than instructions, and set the value of the target register to 1 if the second operand is less than the third, 0 otherwise.

We can use these to implement all of the conditional and looping constructs we are used to in high-level languages.

Suppose `i` is in `$s0`, `j` is in `$s1`, and `h` is in `$s3`.

```
if (i==j) h = i + j;
```

MIPS assembly:

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:  ...
```

Slightly more complex:

```
if (i!=j) h = i + j;
else h = i - j;
```

assembles to:

```
        beq $s0, $s1, ElsePart
        add $s3, $s0, $s1
        j OverIf
ElsePart:
        sub $s3, $s0, $s1
OverIf: ...
```

And an inequality:

```
if (i<j) h = i + j;
else h = i - j;
```

assembles to:

```
        slt $t0, $s0, $s1
        beq $t0, $zero, ElsePart
        add $s3, $s0, $s1
        j OverIf
ElsePart:
        sub $s3, $s0, $s1
OverIf: ...
```

Larger Constants in MIPS

So far, we have seen how to get 16-bit constants to use in immediate mode instructions. But what if we want a 32-bit constant?

MIPS requires that all instructions fit in a single 32-bit word, so we can't have an opcode and the whole 32-bit constant at once.

It takes two instructions:

First, "load upper immediate":

```
        lui $t0, 0xa532
```

This sets `$t0` to `0xa5320000`. It is an I-format instruction, using the `address` field of that format to get the 16 bits for the top half of the specified register.

We can then put in appropriate lower order bits:

```
      ori $t0, 0x8d7e
```

This will "or in" the bottom bits to have the constant specified, leaving the upper bits that we've already set alone. `$t0` is now `0xa5328d7e`.

What Else is Missing?

The MIPS ISA doesn't provide instructions for operations that can easily be expressed as an existing operation.

For example, you might want to copy a value in one register to another:

```
        move $t0, $t1
```

This is valid MIPS *assembly* language, but not valid MIPS *machine* language. This is a *pseudoinstruction*.

As assembler would encode this as:

```
        add $t0, $t1, $zero
```

In this case, there's no extra cost. It's still just one instruction.

Other pseudoinstructions may translate to more than one instruction. For example, the pseudoinstruction `bgt`, which branches on greater than:

```
        bgt $s1, $s2, Label
```

would likely translate to

```
        slt $at, $s2, $s1
        bne $at, $zero, Label
```

Note the use of the "reserved for assembler use" register `$at`.

Others:

- `li` – load immediate
- `la` – load address
- `sgt`, `sle`, `sge` – set on ...
- `bge`, `bgt`, `ble`, `blt` – conditional branches

When determining relative costs of different translations of high-level language into assembly, this pseudoinstruction should be considered to cost twice as much as a regular instruction or a pseudoinstruction that corresponds directly to a single machine instruction.

The text goes into more detail about the MIPS ISA, including the mechanisms for procedure calls, I/O, and more. We will return to some of this later.