Extended Assembler

Machine language: very low level
Assembler: provides higher-level language for convenience in programming

Register mnemonics

- We've already used them. The real machine deals in register numbers (0-31).
- Only $0 and $31 are special in the hardware.
- Other registers are used for particular purposes by software convention.

Pseudoinstructions

- Instructions or formats which are not directly implemented in the hardware.
- CISC would include many alternative forms of instructions.
  - Large and slow instruction sets
- Pseudoinstruction may be translated to 1 or more real instructions.
- Pseudocomputer: more flexible than real computer, easier to program
  - Another layer of abstraction

Labels

- Can use identifiers (names) to represent locations in the program
- Assembler calculates necessary offsets

Directives

- Control layout and processing of program

Pseudoinstructions: Data transfer (register)

Instruction	Real instructions	Semantics
Copy contents of register s to register t		
`mov $rt, $rs`	`addi $rt, $rs, 0`	`R[t] = R[s]`
Load immediate into register s		
`li $rs, immed`		`R[s] = immed`

The way this is translated depends on whether immed is 16 bits or 32 bits:

`li $rs, small`	`ori $rs, $0, small`	`R[s] = small`
`li $rs, -small`	`addiu $rs, $0, -small`	`R[s] = -small`
`li $rs, big`	`lui $rs, upper(big)` `ori $rs, $rs, lower(big)`	`R[s] = big`

small: 16-bit value

big: 32-bit value

Note: `upper(big)` and `lower(big)` are not real instruction syntax

The assembler must figure out how to get the upper 16 bits of a 32-bit value:

upper (big) = big_{31-16} lower (big) = big_{15-0}

Load address into register s

`la $rs, addr`	`lui $rs, upper(addr)` `ori $rs, $rs, lower(addr)`	`R[s] = addr`

Pseudoinstructions: Data transfer (memory)

Load a word into memory with a 32-bit offset (called big).
Notice that this is normally not allowed, because only 16-bit offsets are permitted.

Instruction	Real instructions	Semantics
`lw $rt, big($rs)`	`lui $at, upper( big )`	`Addr <-- R[s] + big`
	`ori $at, $at, lower( big )`	`R[t] <--` M_4`[ Addr ]`
	`add $at, $rs, $at`	
	`lw $rt, 0($at)`	

Similar pseudo-instructions exist for sw, etc.
Other size load, store:

`ld, sd`	**doubleword**
`ulh, ulw, ush, usw`	**unaligned halfword, word**

Pseudoinstructions: Branch

How do we compare values in 2 registers?
Instructions for beq, bne, but not for general relational operators

		result
`slt $rd, $rs, $rt`	`R[s] <  R[t]`	1
	`R[s] >= R[t]`	0

Instruction	Real instructions	Semantics
`bge $rs, $rt, LABEL`	`slt $at, $rs, $rt` `beq $at, $zero, LABEL`	`if (R[s] >= R[t])` `goto LABEL`
`bgt $rs, $rt, LABEL`	`slt $at, $rt, $rs` `bne $at, $zero, LABEL`	`if (R[s] > R[t])` `goto LABEL`
`ble $rs, $rt, LABEL`	`slt $at, $rt, $rs` `beq $at, $zero, LABEL`	`if (R[s] <= R[t])` `goto LABEL`
`blt $rs, $rt, LABEL`	`slt $at, $rs, $rt` `bne $at, $zero, LABEL`	`if (R[s] < R[t])` `goto LABEL`

Note that LABEL must be converted to an offset from PC

What about immediate value?

```
bge $rs, immed, LABEL
```

Pseudoinstructions: Branch

Comparison to 0

Instruction	Real instructions	Semantics
`beqz $rs, LABEL`	`beq $rs,$zero,label`	`if (R[s] == 0)` `goto LABEL`
`bnez $rs, LABEL`	`bne $rs,$zero,label`	`if (R[s] != 0)` `goto LABEL`

Pseudoinstructions: Arithmetic

Instruction	Real instructions	Semantics
Multiply		
`mul $rd, $rs, $rt`	`multu $rs, $rt` `mflo $rd`	`# R[d] = R[s] * R[t]`
Multiply with overflow		

`mulo $rd, $rs, $rt`	`mult $rs, $rt`
	`mflo $rd`
	`# check for overflow`
		`# R[d] = R[s] * R[t]`

Pseudoinstructions: Set

Instruction | **Real instructions** | **Semantics**

Set if equal:

```
seq $rd, $rs, $rt    andi $rd, $rd, 0        # R[d] = (R[s] == R[t]) ? 1 : 0
                     bne $rs, $rt, next
                     ori $rd, $zero, 1
                next:
```

What's wrong with this?

Better way:

```
                     xor $rd, $rs, $rt       # R[d] = ~(R[s] == R[t])
                     sltiu $rd, $rd, 1       # R[d] = (R[d] < 1)
```

Set if not equal:

```
sne $rd, $rs, $rt                            # R[d] = (R[s] != R[t]) ? 1 : 0
                     xor $rd, $rs, $rt       # R[d] = ~(R[s] == R[t])
                     sltu $rd, $0, $rd       # R[d] = (R[d] > 0)
```

Set if greater than or equal:

```
sge $rd, $rs, $rt                            # R[d] = (R[s] >= R[t]) ? 1 : 0
                     slt $rd, $rs, $rt       # R[d] = (R[s] < R[t]) ? 1 : 0
                     xori $rd, $rd, 1        # R[d] = ~R[d]
```

Other combinations, including unsigned:

```
sgeu, sgt, sgtu, sle, sleu
```

Pseudoinstructions: logical

Instruction	Real instructions	Semantics
`not $rd, $rs`	`addi $at, $0, -1`	`# R[1] = -1`
	`xor $rd, $rs, $at`	`# R[d] = R[s] ^ R[1]`

Better way:

`not $rd, $rs`	`nor $rd, $rs, $0`	`# R[d] = ~R[s]`

Why does this work?

a	b	a | b	~(a|b)	~(0|b)	~b
0	0	0	1	1	1
0	1	1	0	0	0
1	0	1	0		
1	1	1	0		

Pseudoinstructions: summary

Data transfer	Register	`mov`
	Constant	`li`
	Address	`la`
	Big offset	`lw`
Branch	Greater than or equal	`bge`
	Greater than	`bgt`
	Less than or equal	`ble`
	Less than	`blt`
	Equal 0	`beqz`
	Not equal 0	`bnez`
Set	Equal	`seq`
	Not equal	`sne`
	Greater than or equal	`sge`
	Greater than	`sgt`
	Less than or equal	`sle`
Arithmetic	Multiply	`mul`
	Multiply (overflow)	`mulo`
Logical	Complement	`not`

Extended Assembler

```
## Program to add two plus three
        .text
        .globl  main

main:
        ori     $8,$0,0x2       # put two's comp. two into register 8
        ori     $9,$0,0x3       # put two's comp. three into register 9
        addu    $10,$8,$9       # add register 8 and 9, put result in 10

## End of file
```

Directives

`.text` **defines beginning of source code**

`.globl` **identifies global label**

Label (symbolic address)

`main`

Defining data

```
      .data                          # defines beginning of data area
arr:  .word     2, 4, 6              # defines array of 3 words (int)
chr:  .byte     65                   # defines 1 byte (char)
str:  .asciiz  "a string"            # defines a C-type character string
```