PIEROGI

ISA definition + architecture description

- 32b instructions (all the same length)

32b				
4b	4b	4b	4b	16b
Op	**Rd**	**Ra**	**Rb**	**Imm_{16}**

- Von Neuman architecture
- Four-cycle
- Memory mapped IO

Register table

Number	ASM notation	Description
$0	$zero	Always equals 0
$1	$at	Assembler reserved
$2	$v	Return value
$3 - $5	$a0 - $a2	Arguments
$6 - $8	$s0 - $s2	General purpose (preserved across function calls)
$9 - $12	$t0 - $t3	General purpose (NOT preserved across function calls)
$13	$gv	Global variable pointer
$14	$ra	Return address
$15	$sp	Stack pointer

Instruction table

Opcode		ASM notation	Description
Bitwise instructions			
0x0	0000	**and** $rd, $ra, $rb	Rd = Ra & Rb
0x1	0001	**or** $rd, $ra, $rb	Rd = Ra | Rb
0x2	0010	**xor** $rd, $ra, $rb	Rd = Ra ^ Rb
0x3	0011	**not** $rd, $ra	Rd = ~Ra
Arithmetic instructions			
0x4	0100	**add** $rd, $ra, $rb	Rd = Ra + Rb
0x5	0101	**sub** $rd, $ra, $rb	Rd = Ra - Rb
0x6	0110	**cmp** $rd, $ra, $rb	Rd = (Ra>Rb)
Jumps			
0x7	0111	**j** $rd, $ra	{ Rd = PC + 4; PC = Ra }
0x8	1000	**beq** $ra, $rb, Imm_{16}	if(Ra == Rb){ PC = PC + Imm_{16} }
0x9	1001	**bne** $ra, $rb, Imm_{16}	if(Ra != Rb){ PC = PC + Imm_{16} }
Arithmetic immediate instructions			
0xA	1010	**sl** $rd, $ra, Imm_{16}	Rd = Ra << Imm_{16}
0xB	1011	**sr** $rd, $ra, Imm_{16}	Rd = Ra >> Imm_{16} (logical)
0xC	1100	**addi** $rd, $ra, Imm_{16}	Rd = Ra + Imm_{16}
0xD	1101	**lui** $rd, Imm_{16}	Rd = (Imm << 16)
Memory access			
0xE	1110	**lw** $rd, $rb	Rd = word stored in memory at (Rb)
0xF	1111	**sw** $ra, $rb	word stored in memory at (Rb) = Ra

Pseudo-instructions

ASM notation	Description	Translates to
la $rd, identifier	Load address of identifier into Rd	**lui** $at, address[31:16] **addi** $rd, $0, address[15:0] **sl** $rd, $rd, 16 **sr** $rd, $rd, 16 **add** $rd, $at, $rd
la $rd, Imm_{32}	Load address in Imm_{32} into Rd	
ja $rd, identifier	Jump to address of identifier unconditionally	**lui** $at, address[31:16] **addi** $rd, $0, address[15:0] **sl** $rd, $rd, 16 **sr** $rd, $rd, 16 **add** $at, $at, $rd **j** $rd, $at
ja $rd, Imm_{32}	Jump to address given by Imm_{32}	
beq $ra, $rb, identifier	Jump to address of identifier on equal	**beq** $ra, $rb, Diff (difference between address of beq and identifier)
bne $ra, $rb, identifier	Jump to address of identifier on unequal	**bne** $ra, $rb, Diff (difference between address of beq and identifier)
push $ra	Push the contents of $ra onto the stack	**addi** $sp, $sp, -4 **sw** $ra, $sp
pop $rd	Pop the top value on the stack into $rd	**lw** $rd, $sp **addi** $sp, $sp, 4
identifier:	Sets the identifier to the address of the next instruction / block	

Directives

Name	Purpose
.data	Beginning of the data segment
.text	Beginning of the text segment
.word	Word segment - marks and assigns a value to a one word variable in memory
.space	Space segment - marks a contiguous block of memory and zeroes it
.addr	Address segment - marks a specific address

Summary of all supported instructions

Instruction	Name / Mnemonic
`and`	(bitwise, logical) AND
`or`	(bitwise, logical) OR
`xor`	(bitwise, logical) XOR
`not`	(bitwise, logical) NOT
`add`	(signed) addition
`sub`	(signed) subtraction
`cmp`	(unsigned) comparison
`j`	jump, storing return address
`beq`	branch on equal
`bne`	branch on unequal
`sl`	(logical) shift left
`sr`	(logical) shift right
`addi`	(signed) addition with immediate
`lui`	load upper immediate, load immediate into upper halfword
`lw`	load word
`sw`	store word
`la`	load address
`ja`	jump to address, storing return address
`push`	push onto stack
`pop`	pop from stack

Memory map

Address range	Purpose
`0x00000000 - 0x0fffffff`	Text segment (instructions), growing **up**
`0x10000000 - 0x7fffffff`	Data segment (memory), growing **up**
`0x7fffffff - 0x10000000`	Stack, growing **down**
`0x80000000 - 0xfffeffff`	Reserved
`0xffff0000 - 0xffffffff`	IO segment (memory mapped IO)

Memory mapped IO

Address	Size	R/W	Peripheral
`0xffff0000`	`1 word`	`R`	Keyboard (returns ASCII value of pressed key)
`0xffff0004`	`1 word`	`W`	Decimal display (treats input as signed, 32b)
`0xffff0008`	`1 word`	`W`	Reserved for display configuration

Writes into read-only registers are ignored. Reads from write-only registers return 0.

Assembly program structure

Files

A PIEROGI assembly program consists of an arbitrary number of files. During assembly, files are concatenated and processed together.

Segments

A file consists of one or more segments. The segments can go in any order. All code must be inside a segment. There are two types of segments:
.data - Contains data blocks
.text - Contains instructions

Blocks

A .data segment consists of blocks, each marking a location in memory. There are three block types:
.word - marks a single word holding a pre-assigned value
.space - marks a contiguous range of zero-initialized words
.addr - marks an address without claiming space or initializing it, which can be useful for memory mapped IO.

Identifiers

Any instruction or memory block can be annotated with an identifier, which can be any string consisting of upper or lower case ASCII letters and underscores. This identifier can be used in pseudo-instructions. An identifier is defined by it's name with a semicolon ("main:") and can then be used (without a semicolon) as a pseudo-instruction argument. An identifier in use has to be defined exactly once, but it can be defined anywhere (definition does not have to precede the first use).

The "main" identifier marks the entry point of the program and is required to be present.

(Pseudo-)Instructions

An instruction consists of an instruction name and a comma-separated list of registers; for some instructions, this can include immediate or identifier.

Comments

Comments are regions ignored by the assembler. They start with '#' and end with a newline (or end of file).

Assembly program example

```
.data # all blocks have to be inside a .data segment
    constant: .word 5
    # “constant” points to 1 word holding the value 5
    # with automatically assigned address
    buffer: .space 3
    # “buffer” points to the first of three words, init. to zero,
    # with automatically assigned address
    peripheral: .addr 0xffff0004
    # “peripheral” points to address 0xffff0004, uninitialized.
    # The memory location could be later assigned to some other block
    # by the assembler and the reference would remain valid.
.text # all instructions have to be inside a .text segment
main: # execution starts at the main function
    la $s1, constant
    # la loads the address of an “constant” to a ragister

    addi $s0, $zero, 5 # a simple “for” loop with 5 iterations
    loop_start: # loop_start will be assigned to the next address
    beq $s0, $zero, loop_end
        addi $a0, $s0, 0 addi $a1, $s1, 0 # prepare function arguments
        ja $ra, func_add # function call
        addi $s1, $v, 0  # save function output
        addi $s0, $s0, -1
    beq $zero, $zero, loop_start
    loop_end:
    add $t1, $zero, $zero # if this last instruction was not here,
    # loop_end would have no address assigned and would be ignored

func_add:
# this is a simple function that adds its operands
# function arguments go into $a0 and $a1, $v is returned
# functions are allowed to overwrite the contents of $a*, $t* and $v
    push $ra # store the return address on the stack, which is
             # required if the functions calls other functions
    add $v, $a0, $a1 # perform any internal logic
    pop $ra  # retrieve return address; be careful about memory leaks
    j $zero, $ra # return
```

GREG

The reference assembler

Features

- Supports decimal and hexadecimal immediate
- Pseudo-instruction support
  - Memory location labels
  - 32b immediate operations
  - Stack operations
- Supports linking multiple source files
- Automatic prelude insertion
  - Initializes the stack and global pointer
  - Jumps to the main function
- Binary or Intel HEX output
- Helpful error messages
  - Kind of error
  - Affected file and location in file
- Implemented as a Rust library
  - API for integration into other projects
  - High level language, modular
    -> easily extensible
  - GPLv3

GET HERE

Known limitations

- There can only be one label per address, not counting .addr blocks
- Immediate is not checked for size
  - Values are truncated when too wide
- Negative hexadecimal immediate values are not supported
- Byte operations are not supported
  - Accessing unaligned memory is undefined