The *pico RISC* Processor Architecture and Assembly

Principles

- A 32-bit RISC processor architecture, extremely simplified for educational purposes
- Only the very elementary architectural elements for basic understanding, esp. for programming low-level system software
- Load/store RISC orthogonal architecture
- 32-bit (virtual) address space, byte as the addressable unit, little-endian format
- three-address integer arithmetic only (no floating point), signed or unsigned integer comparison and extension
- Two modes: privileged (kernel, system) and user
- The remaining specifications deals with the user mode only, i.e., with the architecture visible to the compiler and application programmer

Registers

- 16 x 32-bit general-purpose registers: R0..R15
- 32-bit program counter: PC
- 32-bit stack pointer: SP; the stack grows towards lower addresses, SP points to the last pushed byte

Address Modes

- Immediate: #*constant-expression*
- Register direct: R*i*
- Memory direct: *address; address* is a *constant-expression*
- Register indirect: [R*i*]
- Register indirect with displacement: [R*i* + *offset*]; *offset* is a *constant-expression*
- Registers usable in address modes: R0..R15, PC, SP
- Address modes usage:
    - in Control Flow instructions: all memory modes (memory direct, register indirect, register indirect with displacement)

    - in Load/Store instructions: all (except immediate for store)
    - in Stack Instructions and Arithmetic & Logic Instructions: register direct
- Format of instructions:
    - using register direct or register indirect modes:

31	23	20	15	10	5	2
Op code	Addr Mode	Reg0	Reg1	Reg2	Type	Unused

    - using immediate, memory direct, or register indirect with displacement modes:
        - First double-word (first 32 bits):

31	23	20	15	10	5	2
Op code	Addr Mode	Reg0	Reg1	Unused	Type	Unused

        - Second double-word (second 32 bits):

31
Constant/Address/Displacement

    - Address Mode Codes:
        - Immediate: 0b100
        - Register direct: 0b000
        - Memory direct: 0b110
        - Register indirect: 0b010
        - Register indirect with displacement: 0b111
    - Register codes:
        - R0..R15: 0x0..0xf
        - SP: 0x10
        - PC: 0x11
    - Data type codes:
        - double-word (32 bits): 0b000
        - word (16 bits) – zero extend: 0b001

        - word (16 bits) – sign extend: 0b101
        - byte (8 bits) – zero extend: 0b011
        - byte (8 bits) – sign extend: 0b111

Instructions

Control Flow Instructions

- The address of the target instruction is obtained as the address of the “operand” as specified in the (memory) address mode
- Format:
    - using register direct or register indirect modes:

31	23	20	15	10
Op code	Addr Mode	Reg0	Reg1/Unused	Unused

    - using memory direct or register indirect with displacement modes:
        - First double-word (first 32 bits):

31	23	20	15	10
Op code	Addr Mode	Reg0	Reg1/Unused	Unused

        - Second double-word (second 32 bits):

31
Address/Displacement

- Instructions:

Instruction	*Op code*	*Address Modes*	*Comment*
INT op	0x00	Register direct (exception)	Software interrupt/system call; the Reg0 code is used as the interrupt number
JMP op	0x02	memory direct, register indirect, register indirect w/ displ	Absolute or relative jump
CALL op	0x03	-‖-	Call subroutine. PC pushed on stack

RET	0x01	None	Return from subroutine. PC popped from the stack
JZ reg1, op	0x04	For op: memory direct, register indirect, register indirect w/ displ For reg1: register direct	If Reg1==0, jump to op
JNZ reg1, op	0x05	-‖-	If Reg1!=0, jump to op
JGZ reg1, op	0x06	-‖-	If Reg1>0 (as signed int), jump to op
JGEZ reg1, op	0x07	-‖-	If Reg1>=0 (as signed int), jump to op
JLZ reg1, op	0x08	-‖-	If Reg1<0 (as signed int), jump to op
JLEZ reg1, op	0x09	-‖-	If Reg1<=0 (as signed int), jump to op

Load/Store Instructions

- Load: copy the value from a memory location to a register
- Store: copy the value from a register to a memory location
- Loading from memory to a register – types of source:
  - byte, zero-extended or sign-extended to a register; instruction mnemonic suffix: UB (unsigned) or SB (signed)
  - word (16 bits), zero-extended or sign-extended to a register; instruction mnemonic suffix: UW (unsigned) or SW (signed)
  - double-word (32 bits); no instruction mnemonic suffix
- Storing from a register to memory – types of transfer:
  - byte (least-significant byte of register); instruction mnemonic suffix: B
  - word (lower 16 bits of register); instruction mnemonic suffix: W
  - double-word (entire register)
- Format:
  - using register direct or register indirect modes:

31	23	20	15	10	5	2
Op code	Addr Mode	Reg0	Reg1	Unused	Type	Unused

- using memory direct or register indirect with displacement modes:
  - First double-word (first 32 bits):

31	23	20	15	10	5	2
Op code	Addr Mode	Reg0/Unused	Reg1	Unused	Type	Unused

  - Second double-word (second 32 bits):

31
Address/Displacement

- Instructions:

Instruction	*Op code*	*Address Modes*	*Comment*
LOAD reg1, op	0x20	all	Load
STORE reg1, op	0x21	all except immediate	Store

Stack Instructions

- Push or pop registers to/from the stack
- The stack grows towards lower addresses, SP points to the last pushed byte
- These instructions change SP implicitly
- Register direct mode only
- The entire 32-bit register is pushed always (4 bytes)
- Format:

31	23	20	15
Op code	Addr Mode = 0b000	Reg0	Unused

- Instructions:

Instruction	*Op code*	*Address Modes*	*Comment*
PUSH reg	0x22	Register direct	Push register
POP reg	0x23	Register direct	Pop register

Arithmetic and Logic Instructions

- Operations with entire 32-bit registers only
- Three-address operations, register direct mode only
- Signed integer arithmetic
- Format:

31	23	20	15	10	5
Op code	Addr Mode = 0b000	Reg0	Reg1	Reg2	Unused

- Instructions:

Instruction	*Op code*	*Address Modes*	*Comment*
ADD reg0, reg1, reg2	0x30	Register direct	Reg0 = Reg1 + Reg2
SUB reg0, reg1, reg2	0x31	-||-	Reg0 = Reg1 - Reg2
MUL reg0, reg1, reg2	0x32	-||-	Reg0 = Reg1 * Reg2
DIV reg0, reg1, reg2	0x33	-||-	Reg0 = Reg1 / Reg2
MOD reg0, reg1, reg2	0x34	-||-	Reg0 = Reg1 % Reg2
AND reg0, reg1, reg2	0x35	-||-	Reg0 = Reg1 & Reg2
OR reg0, reg1, reg2	0x36	-||-	Reg0 = Reg1 | Reg2
XOR reg0, reg1, reg2	0x37	-||-	Reg0 = Reg1 ^ Reg2
NOT reg0, reg1	0x38	-||-	Reg0 = ˜ Reg1
ASL reg0, reg1, reg2	0x39	-||-	Reg0 = Reg1 << Reg2
ASR reg0, reg1, reg2	0x3A	-||-	Reg0 = Reg1 >> Reg2

Assembly

- Assembly statements:
    - instructions
    - data definitions
    - directives
- Format of instructions:

    label: *op* *oper1, oper2, oper3* *;* *comment*
- *label* and *; comment* are optional
- *label* can be used in *constant-expressions* and resolves into the absolute address of the labeled statement (instruction or data definition)

Data definitions

- Format of definitions:

    label: *def* *data-specifier, ...* *;* *comment*
- *label* and *; comment* are optional
- Definitions:
    - DB: define byte (each data item is of a byte size)
    - DW: define word (each data item is of a 16-bit word size)
    - DD: define double-word (each data item is of a 32-bit double-word size)
- Data item specifiers:

    constant-expression [DUP *constant-expression* | ?]
    - *constant-epxression:* literals, identifiers (defined labels or symbols), integer arithmetic operators (+, -, *, /), subexpressions (parentheses)
    - *literal*: a signed integer literal in decimal, binary, or hexadecimal format (prefix specifies the format as in C) or character (in C format)
    - ? for undefined
    - DUP: repeated definition of data items of the same type (size) as specified by the definition, all initialized as specified by the second expression

Directives

- Define a symbol (a symbolic constant) that can be used in *constant-expressions*:

```
symbol    DEF   constant-expression       ;     comment
```

- An implicitly defined symbol $ resolves into the absolute (starting) address of the current instruction (i.e., the address of the first byte of the instruction where $ is used)
- Define the starting absolute address of the next statement (override $):

```
          ORG   constant-expression       ;     comment
```

- Define the address of the starting instruction (program entry point):

```
          START constant-expression       ;     comment
```

- Define a memory segment:
  - segment begin:

```
          SEG   rwx                       ;     comment
```

  - typically immediately followed by an ORG directive to specify the segment starting address; if ORG is not present, the segment starts at the implicitly assumed address (following the previous statement), possibly aligned to the beginning of the memory allocation unit (e.g. page or segment), depending on the virtual memory organization; the size of the segment is determined by the size of its contents, possibly rounded to a unit of allocation (e.g. page), depending on the virtual memory organization
  - *rwx*: optional specification of access rights; a triplet o binary digits (0 or 1) for *read* (read-only data segment), *write* (write-enabled data segment) and *execute* (read-only code segment, readable during instruction fetch phase only)
  - segment end:

```
          END                             ;     comment
```

Compilation, Linking, and Assembler

- The compiler compiles the picoC++ source into assembly, directly and one-to-one, without any implicit products:
  - no START directive; functions (including *main*) are translated as ordinary subroutines in assembly;
  - no program prologue and epilogue (the code enclosing the call of *main*), no implicit process termination system call;
  - no SEG or ORG directives; the program is translated into a linear sequence of assembly statements, using symbolic absolute addresses (labels) and relative address modes only;
- A separate (quasi-)linker program transforms the assembly generated by the compiler into a processed assembly with:

    - attached program prologue and epilogue (the code enclosing the call of *main*), with an implicit system call to terminate the process (configurable interrupt entry for the system call)
    - the START directive to point to the starting instruction of the prologue;
    - optionally, depending on the configuration:
        - introduces the initial ORG directive to relocate the entire program to the given (configurable) origin (starting address);
        - creates segments (SEG/END directives) by finding the continuous sequences of instructions or data definitions in the source assembly, to create segments out of them; in addition, optionally (depending on the configuration):
            - aligns the segment to the beginning of the memory allocation unit (configurable page/segment size)
            - relocates and merges code segments and data segments
            - creates a stack segment with a configurable size
            - creates a heap (data free store) segment with a configurable size
- A separate assembler program transforms the assembly generated by the linker into the binary code; works in two modes:
    - generate symbolic binary with assembly: generates a textual file with the source assembly code incorporated, along with the translation; format of a statement:

addr byte byte ... byte ; label: op oper1, oper2, oper3 ; comment

        - *addr*: 8 hex digits (as ASCII characters for digits) for the absolute address
        - *byte*: 2 hex digits (as ASCII characters for digits) for the binary translation of the statement
    - generate pure binary executable.
- The executable binary format of the file (little endian integers):

Offset	*Size (bytes)*	*Field*	*Purpose*
0x00	4		0x7F followed by EXE in ASCII; these four bytes constitute the 'magic number'
0x04	1		Identifies the target operating system ABI (type and version)
0x05	11		Unused (reserved for future use)
0x10	4		This is the memory address of the entry point from

			where the process starts executing.
0x14	4		Points to the start of the segment table; it usually follows the file header immediately, making the offset 0x1c
0x18	4		Contains the number of entries in the segment table
0x1c			End (size) of header

- One segment table entry:

Offset	*Size (bytes)*	*Field*	*Purpose*
0x00	4		Type of segment, including *rwx* bits (least significant bits)
0x04	4		Offset of the segment in the file image
0x08	4		Virtual address of the segment in memory
0x0c	4		On systems where physical address is relevant, reserved for segment's physical address
0x10	4		Size in bytes of the segment in the file image; may be 0
0x14	4		Size in bytes of the segment in memory; may be 0
0x18			End (size) of entry