

RISC INSTRUCTION SET ARCHITECTURE: INTEL NIOS 2 PROC.

A PROCESSOR IS DEFINED BY Instructions and Sequencing

- **Instruction Set Architecture (ISA)** can be seen as the specifications of a processor
 - ISA affects processor performances (RISC, CISC, ASIP*)
 - Possible different implementations of the same ISA
- **Instructions for a computer must support:** *
 - data transfers to and from the memory
 - arithmetic and logic operations on data
 - program sequencing and control
 - input/output transfers

* Application-Specific Instruction set Processor

the Instruction set is a
Tradeoff between:
- performance
- complexity
*BUT MUST SUPPORT: **


RISC and CISC Instruction Sets

- Nature of instructions distinguishes computer
- Two fundamentally different approaches

ARM – Reduced Instruction Set Computers (RISC) have one-word instructions and require arithmetic operands to be in registers

x86 – Complex Instruction Set Computers (CISC) have multi-word instructions and allow operands directly from memory

RISC Instruction Sets

- Each RISC instruction occupies a single word
- A load/store architecture is used, meaning:
 - only Load and Store instructions are used to access memory operands
 - operands for arithmetic/logic instructions must be in registers, or one of them can be provided explicitly in the instruction word (*Immediate* operand)
 - Load *proc_register, mem_location*
 - Addressing mode specifies actual memory location

Nios II Main Characteristics

- RISC-style architecture (**all instructions are 32-bit long**)
- 32-bit data *word*
- 2x memory interfaces (Harvard architecture)
- **Byte-addressable** memory space:
 - **With little-endian addressing scheme**
(lower byte addresses used for less significant bytes)
 - **The LOAD and STORE instructions can transfer data in:**
word, half-word, and byte
- 32 general-purpose **registers, 32-bit long**
- **Several additional control registers**
- **2 versions: economy (5-stage) and fast (6-stage w. pipelining)**

Nios II registers (1)

CONVENTION OF COMPILER

- General-purpose registers (r0-r31)

Register	Name	Function	Register	Name	Function
r0	zero	0x00000000	r16		Callee-saved register
r1	at	Assembler temporary	r17		Callee-saved register
r2		Return value	r18		Callee-saved register
r3		Return value	r19		Callee-saved register
r4		Register arguments	r20		Callee-saved register
r5		Register arguments	r21		Callee-saved register
r6		Register arguments	r22		Callee-saved register
r7		Register arguments	r23		Callee-saved register
r8		Caller-saved register	r24	et	Exception temporary
r9		Caller-saved register	r25	bt	Breakpoint temporary (1)
r10		Caller-saved register	r26	gp	Global pointer
r11		Caller-saved register	r27	sp	Stack pointer
r12		Caller-saved register	r28	fp	Frame pointer
r13		Caller-saved register	r29	ea	Exception return address
r14		Caller-saved register	r30	ba	Breakpoint return address (2)
r15		Caller-saved register	r31	ra	Return address

Addressing Modes (1)

- How operands are specified in an instruction
- Nios 2 proc. supports 5 addressing modes:
 - *Immediate mode*: a 16-bit operand is contained in the instruction itself. This value is (sign-)extended to produce a 32-bit operand for (arithmetic) instructions
 - *Register mode*: the operand is the content of a register
 - *Register indirect mode*: the effective address of the operand is the content of a register

$$n = 32$$

$a_{n-1} a_{n-2} \dots a_0$

UNSIGNED
NUMBER

$$2^{n-1} a_{n-1} + 2^{n-2} a_{n-2} + \dots + 2a_1 + a_0$$

SIGNED
NUMBER

(2's complement)

$$-2^{n-1} a_{n-1} + 2^{n-2} a_{n-2} + \dots + 2a_1 + a_0$$

$$a_{n-1} a_{n-1} a_{n-2} \dots a_1 a_0$$

MUST
REPLICATE

MOST SIGNIFICANT
BIT

$$(-2^n a_{n-1} - 2^{n-1} a_{n-1} + 2^{n-1} a_{n-1}) + 2^{n-2} a_{n-2} + \dots + 2a_1 + a_0$$

Addressing Modes (2)

- Nios 2 proc. supports 5 addressing modes:
 - *Displacement mode*: the effective address of the operand is obtained by adding the content of a register and a 16-bit value contained in the instruction itself.
 - *Absolute mode*: is a particular case of the *Displacement mode* when the register is r0
- E.g. `addi r3, r2, 100`
the content of r2 is added to 100 and the result placed in r3

Addressing Modes (3)

Involves appropriate extension to 32 bit

Nios II addressing modes.

Name	Assembler syntax	Addressing function
Immediate	Value	Operand = Value
Register	ri	$EA = ri$
Register indirect	(ri)	$EA = [ri]$
Displacement	$X(ri)$	$EA = [ri] + X$
Absolute	$LOC(r0)$	$EA = LOC$

EA = effective address

Value = a 16-bit signed number

X = a 16-bit signed displacement value

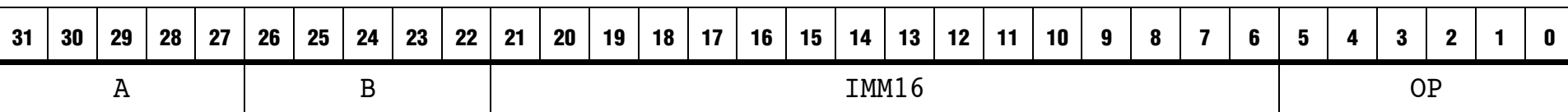
$[ri]$ indicates the content of the register ri

Instruction formats (1)

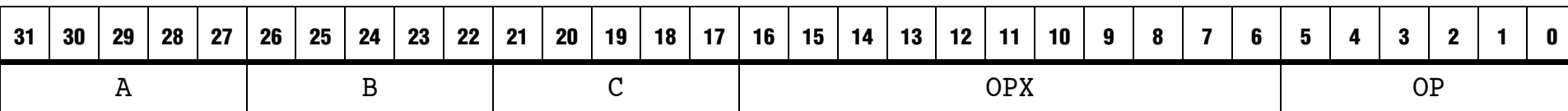
- **RISC-style instructions** (all 32-bit long)
 - **Load/store architecture** for **data transfers**
 - **Arithmetic/logic instructions** use **registers**
- Three instruction types:
 - I-type** OP dst_reg, src_reg, immediate
 - R-type** OP dst_reg, src_reg1, src_reg2
 - J-type** OP label_or_address
- label_or_address is a 26-bit unsigned immediate value

Instruction formats (2)

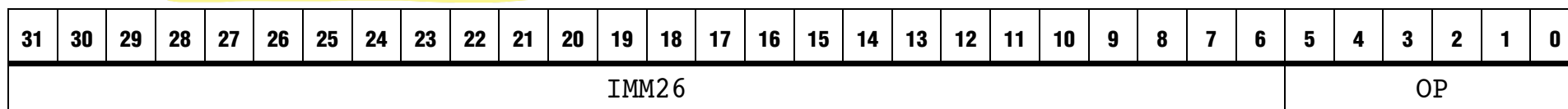
- I-type instructions** include arithmetic and logical operations in which one operand is a constant, such as **addi** and **andi**; **branch** operations; **load** and **store** operations; and cache management operations.



- R-type instructions** include arithmetic and logical operations such as **add**, **and**, **nor**; comparison operations such as **cmpeq** and **cmplt**



- J-type instructions** such as **call** and **jmp**, transfer execution anywhere within a 256-MB range



ADD R4, R3, R0

always "0"
sums R0 and R3 and saves
result in R4

↓ is like doing

MOV R4, R3

copies content from R3 to R4

$R8 \leftarrow (\text{address-high}, \text{address-low})$

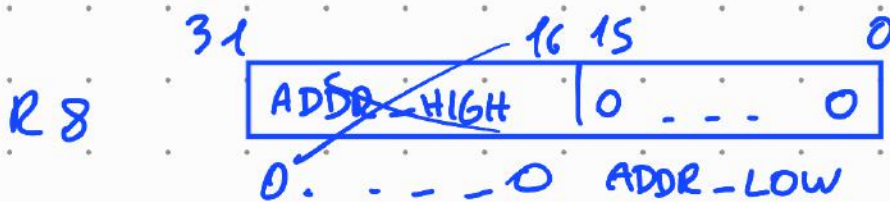
16 BITS

16 BITS

orhi R8, R0, address-high

ori R8, R8, address-low

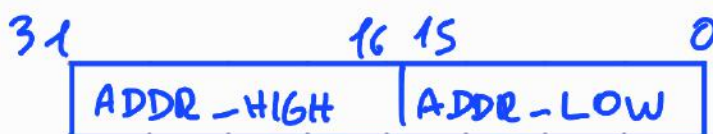
movia R8, ADDRESS



or

↓

R8



STW / STW10

store word to memory / 10

LDW / LDW10

load // peripheral

Notation Conventions

Notation	Meaning
$X \leftarrow Y$	X is written with Y
$PC \leftarrow X$	The program counter (PC) is written with address X; the instruction at X is the next instruction to execute
PC	The address of the assembly instruction in question
rA, rB, rC	One of the 32-bit general-purpose registers
prs.rA	General-purpose register rA in the previous register set
IMM _n	An n-bit immediate value, embedded in the instruction word
IMMED	An immediate value
X_n	The nth bit of X, where n = 0 is the LSB
$X_{n..m}$	Consecutive bits n through m of X
0xNNMM	Hexadecimal notation
$X : Y$	Bitwise concatenation For example, $(0x12 : 0x34) = 0x1234$
$\sigma(X)$	The value of X after being sign-extended to a full register-sized signed integer
$X \gg n$	The value X after being right-shifted n bit positions
$X \ll n$	The value X after being left-shifted n bit positions
$X \& Y$	Bitwise logical AND
$X Y$	Bitwise logical OR
continued...	

Notation Conventions (con't)

Notation	Meaning
$X \wedge Y$	Bitwise logical XOR
$\sim X$	Bitwise logical NOT (one's complement)
Mem8[X]	The byte located in data memory at byte address X
Mem16[X]	The halfword located in data memory at byte address X
Mem32[X]	The word located in data memory at byte address X
label	An address label specified in the assembly file
(signed) rX	The value of rX treated as a signed number
(unsigned) rX	The value of rX treated as an unsigned number

Load and Store Instructions

- For moving data between memory (or I/O) and general-purpose registers
- Words, half-words, bytes; *alignment required*
- Variants available for I/O (uncached) access
- Examples:

```
ldw    r2, 40(r3)    // load word
stb     r6, 4(r12)    // store byte
ldhio  r9, (r20)     // load I/O halfword
                        // signed extended
ldbu   r2, -100(r3)   // load byte zero
                        // extended
stw     r7, 100(r0)   // store word
```


Arithmetic Instructions

- **add, addi** (16-bit immediate is sign-extended)
- **sub, subi, mul, and muli** are similar
- Mult. is unsigned, result is truncated to 32 bits
- **div** (signed values), **divu** (unsigned values)
- Examples:

```
add    r2, r3, r4      // (r2 ← [r3] + [r4])
muli   r6, r7, 4096     // (r6 ← [r7] × 4096)
divu   r8, r9, r10      // (r8 ← [r9] / [r10])
```

Logic Instructions

- **and, or, xor, nor** have 2 register operands
- **andi, ori, xori, nori** have a register operand and an immediate operand that is **zero-extended** from 16 bits to 32 bits

- Examples:

```
or    r7, r8, r9      // (r7 ← [r8] OR [r9])
```

```
andi  r4, r5, 0xFF    // (r4 ← [r5] AND 255)
```

- **andhi, orhi, xorhi** shift 16-bit immediate left and clear lower 16 bits to zero

Move Pseudo-Instructions

- Pseudoinstructions provided for convenience:

```
mov    ri, rj      =>  add    ri, r0, rj
movi   ri, Val16    =>  addi   ri, r0, Val16
moviu  ri, Val16    =>  ori    ri, r0, Val16
```

- Move Immediate Address for 32-bit value:

```
movia  ri, LABEL    =>  orhi   ri, r0, LABEL_HI
                                ori   ri, ri, LABEL_LO
```

- LABEL_HI is upper 16 bits of LABEL, and
LABEL_LO is lower 16 bits of LABEL

Branch and Jump Instructions

- Unconditional branch: **br LABEL**
 - Instruction encoding uses **signed 16-bit byte offset**
 - Signed/unsigned comparison and branch:
- **beq, bne, bge, bgeu, bgt, bgtu, ble, and bleu**
 - Unconditional branch beyond 16-bit offset:

```
blt  ri,  rj, LABEL    // signed [ri]<[rj]
```

```
bltu ri,  rj, LABEL    // unsigned [ri]<[rj]
```

```
jmp  ri    // jump to address in ri
```

Subroutine Linkage Instructions

- Subroutine **call** instruction: **call LABEL**
- Saves return address (from PC) in **r31 (ra)**
- Target encoded as 26-bit immediate, **Value26**
- At execution time, 32-bit address derived as:
Jump address: **PC₃₁₋₂₈ : Value26 : 00**
- **Call** with target in register: **callr ri**
- **Return** instruction: **ret**
 - Branches to address saved in **r31 (ra)**

Comparison Instructions

- Result of comparing two operands is placed in destination register: 1 (if true) or 0 (if false)

- Less-than comparisons that set ri to 0 or 1:

```
cmplt   $ri$ ,  $rj$ ,  $rk$     // signed  $[rj] < [rk]$   
cmpltu  $ri$ ,  $rj$ ,  $rk$     // unsigned  $[rj] < [rk]$   
cmplti  $ri$ ,  $rj$ , Val16 // signed  $[rj] < \text{Val16}$   
cmpltui  $ri$ ,  $rj$ , Val16 // unsigned  $[rj] < \text{Val16}$ 
```

- Val16 is sign- or zero-extended based on type
- Similarly for: ...eq.., ...ne.., ...le.., ...ge.., ...gt..

Shift and Rotate Instructions

- Shift right logical **rj**, destination register is **ri**:

```
srl  ri, rj, rk    //shift by amount in rk  
srli ri, rj, Val5  //shift by immediate value
```

- Shift right arithmetic **sra**, **srai**: same as above except that sign in bit rj_{31} is preserved
- Shift left logical **sll**, **slli**
- Rotate left **rol**, **roli**
- Rotate right **ror** (no immediate version)

Pseudoinstructions

- `mov`, `movi`, and `movia` already discussed;
translated to other instructions by assembler
- Subtract immediate is actually add immediate with
negation of constant:
`subi ri, rj, Value16 => addi ri, rj, -Value16`
- Also can swap operands for comparisons:
`bgt ri, rj, LABEL => blt rj, ri, LABEL`
- Awareness of pseudoinstructions is not critical,
except when examining assembled code

Carry/Overflow Detection for Add

- Nios II does not have condition codes (flags)
- Arithmetic performed in same manner for signed and unsigned operands
- Detect carry/overflow needs more instructions
- **Carry Detection** (unsigned operands):
test if unsigned result is less than either one of the operands:

```
add      r4, r2, r3  
cmpltu   r5, r4, r2
```

- Carry bit is in r5

Carry/Overflow Detection for Add

- **Overflow Detection** (signed operands):
compare signs of operands & result
- Use xor, and to check for same operand signs
and different sign for result:

```
add    r4, r2, r3
xor     r5, r4, r2
xor     r6, r4, r3
and     r5, r5, r6
blt     r5, r0, OVERFLOW
```

- Similar checks for subtract carry/overflow

References

- Intel, “Nios II Processor Reference Guide,”
n2cpu-nii5v1gen2-683836-666887-2.pdf
 - 8. Instruction Set Reference
- C. Hamacher, Z. Vranesic, S. Zaky, N. Manjikian
"Computer Organization and Embedded Systems,"
McGraw-Hill International Edition
 - Appendix B: from B.1 to B.10