

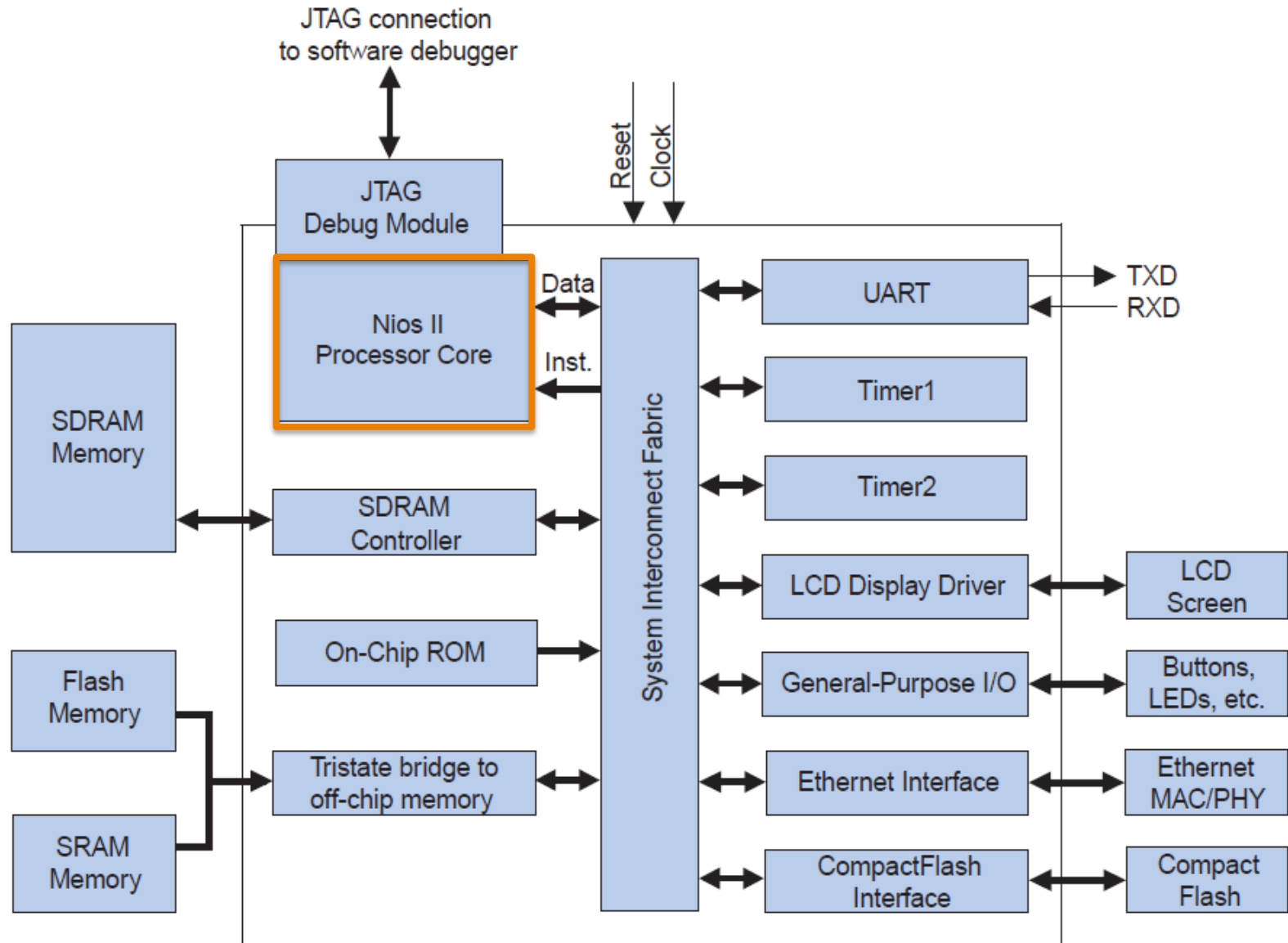
SISTEMI EMBEDDED

AA 2014/2015

Nios II Characteristics
and Architecture

Federico Baronti

Example of a Nios II System (Computer)



Nios II Main Characteristics

- RISC architecture (all instructions are 32-bit long)
- 32-bit data *word*. Data are handled in *word*, *half-word*, and *byte*
- 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

Nios II Other Characteristics (1)

- Optional shadow register sets
- 32 interrupt sources
- External interrupt controller interface for more interrupt sources
- Single-instruction 32×32 multiply and divide producing a 32-bit result
- Dedicated instructions for computing 64-bit and 128-bit products of multiplication

Nios II Other Characteristics (2)

- Floating-point instructions for single-precision floating-point operations
- Single-instruction barrel shifter
- Hardware-assisted debug module enabling processor start, stop, step, and trace under control of the Nios II software development tools
- Optional memory management unit (MMU) to support operating systems that require MMUs

Nios II Characteristics (3)

- Optional memory protection unit (MPU)
- Software development environment based on the GNU C/C++ tool chain and the Nios II Software Build Tools (SBT) for Eclipse
- Integration with Altera's SignalTap[®] II Embedded Logic Analyzer, enabling real-time analysis of instructions and data along with other signals in the FPGA design
- Instruction set architecture (ISA) compatible across all Nios II processor versions
 - Performance up to 250 DMIPS

Nios II Implementation Versions (1)

		economy	standard	fast
Feature		Core		
		Nios II/e	Nios II/s	Nios II/f
Objective		Minimal core size	Small core size	Fast execution speed
Performance	DMIPS/MHz (1)	0.15	0.74	1.16
	Max. DMIPS (2)	31	127	218
	Max. f_{MAX} (2)	200 MHz	165 MHz	185 MHz
Area		< 700 LEs; < 350 ALMs	< 1400 LEs; < 700 ALMs	Without MMU or MPU: < 1800 LEs; < 900 ALMs With MMU: < 3000 LEs; < 1500 ALMs With MPU: < 2400 LEs; < 1200 ALMs
Pipeline		1 stage	5 stages	6 stages
External Address Space		2 GB	2 GB	2 GB without MMU 4 GB with MMU

Nios II Implementation Versions (2)

Feature		Core		
		Nios II/e	Nios II/s	Nios II/f
Instruction Bus	Cache	–	512 bytes to 64 KB	512 bytes to 64 KB
	Pipelined Memory Access	–	Yes	Yes
	Branch Prediction	–	Static	Dynamic
	Tightly-Coupled Memory	–	Optional	Optional
Data Bus	Cache	–	–	512 bytes to 64 KB
	Pipelined Memory Access	–	–	–
	Cache Bypass Methods	–	–	<ul style="list-style-type: none"> ■ I/O instructions ■ Bit-31 cache bypass ■ Optional MMU
	Tightly-Coupled Memory	–	–	Optional
Arithmetic Logic Unit	Hardware Multiply	–	3-cycle (3)	1-cycle (3)
	Hardware Divide	–	Optional	Optional
	Shifter	1 cycle-per-bit	3-cycle shift (3)	1-cycle barrel shifter (3)
JTAG Debug Module	JTAG interface, run control, software breakpoints	Optional	Optional	Optional
	Hardware Breakpoints	–	Optional	Optional
	Off-Chip Trace Buffer	–	Optional	Optional
Memory Management Unit		–	–	Optional
Memory Protection Unit		–	–	Optional

Dhrystone Benchmark (1)

- **Problem: compare processors with very different architectures in a way representative of real-world applications**
 - MIPS are unsuitable to compare RISC with CISC processors, which have very different instruction sets
- Dhrystone benchmark was first published in Ada back to 1984
- Now the C version of Dhrystone is largely used in industry

Dhrystone Benchmark (2)

- Dhrystone compares the performance of the processor under benchmark to that of a **reference machine**
- Dhrystone code dominated by simple integer arithmetic operations, string operations, logic decisions, and memory accesses
- Dhrystone result is determined by measuring the average time a processor takes to perform many iterations of a single loop containing a fixed sequence of instructions that make up the benchmark

Dhrystone MIPS (1)

- The industry has adopted the VAX 11/780 as the reference, namely 1 MIP machine. The VAX 11/780 achieves 1757 Dhrystones per second
- DMIPS figure of a computer is calculated by measuring the number of Dhrystones per second performed by the computer, and dividing it by 1757
 - So "80 DMIPS" means "80 Dhrystone VAX MIPS", which implies 80 times faster than a VAX 11/780
- A DMIPS/MHz rating takes this normalization process one step further, enabling comparison of processor performance at different clock rates

Dhrystone MIPS (2)

- Dhrystone numbers actually reflect the performance of the C compiler and libraries, probably more than the performance of the processor itself. Also, lack of independent certification means that customers are dependent on processor vendors to quote accurate and meaningful Dhrystone data.

“And of course, the very success of a benchmark program is a danger in that people may tune their compilers and/or hardware to it, and with this action make it less useful.”

Reinhold P. Weicker, Siemens AG, April 1989

Author of the Dhrystone Benchmark

Nios II registers (1)

- **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

Nios II registers (2)


- **Control registers**

accessible only by the special instructions *rdctl* and *wrctl* that are only available in supervisor mode

Register	Name	Register Contents
0	status	Refer to Table 3-7 on page 3-12
1	estatus	Refer to Table 3-9 on page 3-14
2	bstatus	Refer to Table 3-10 on page 3-15
3	ienable	Internal interrupt-enable bits (3)
4	ipending	Pending internal interrupt bits (3)
5	cpuid	Unique processor identifier
6	Reserved	Reserved
7	exception	Refer to Table 3-12 on page 3-16
8	pteaddr (1)	Refer to Table 3-13 on page 3-16
9	tlbacc (1)	Refer to Table 3-15 on page 3-17
10	tlbmisc (1)	Refer to Table 3-17 on page 3-18
11	Reserved	Reserved
12	badaddr	Refer to Table 3-19 on page 3-21
13	config (2)	Refer to Table 3-21 on page 3-21
14	mpubase (2)	Refer to Table 3-23 on page 3-22
15	mpuacc (2)	Refer to Table 3-25 on page 3-23
16-31	Reserved	Reserved

Status register (1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								RSIE	NMI	PRS						CRS						IL				IH	EH	U	PIE		

Bit	Description	Access	Reset	Available
RSIE	RSIE is the register set interrupt-enable bit. When set to 1, this bit allows the processor to service external interrupts requesting the register set that is currently in use. When set to 0, this bit disallows servicing of such interrupts.	Read/Write	1	EIC interface and shadow register sets only (4)
NMI	NMI is the nonmaskable interrupt mode bit. The processor sets NMI to 1 when it takes a nonmaskable interrupt.	Read	0	EIC interface only (3)
PRS	<p>PRS is the previous register set field. The processor copies the CRS field to the PRS field upon one of the following events:</p> <ul style="list-style-type: none"> ■ In a processor with no MMU, on any exception ■ In a processor with an MMU, on one of the following: <ul style="list-style-type: none"> ■ Break exception ■ Nonbreak exception when <code>status.EH</code> is zero <p>The processor copies CRS to PRS immediately after copying the <code>status</code> register to <code>estatus</code>, <code>bstatus</code> or <code>sstatus</code>.</p> <p>The number of significant bits in the CRS and PRS fields depends on the number of shadow register sets implemented in the Nios II core. The value of CRS and PRS can range from 0 to $n-1$, where n is the number of implemented register sets. The processor core implements the number of significant bits needed to represent $n-1$. Unused high-order bits are always read as 0, and must be written as 0.</p> <p> Ensure that system software writes only valid register set numbers to the PRS field. Processor behavior is undefined with an unimplemented register set number.</p>	Read/Write	0	Shadow register sets only (3)

Status register (2)

Bit	Description	Access	Reset	Available
CRS	<p>CRS is the current register set field. CRS indicates which register set is currently in use. Register set 0 is the normal register set, while register sets 1 and higher are shadow register sets. The processor sets CRS to zero on any noninterrupt exception.</p> <p>The number of significant bits in the CRS and PRS fields depends on the number of shadow register sets implemented in the Nios II core. Unused high-order bits are always read as 0, and must be written as 0.</p>	Read (1)	0	Shadow register sets only (3)
IL	IL is the interrupt level field. The IL field controls what level of external maskable interrupts can be serviced. The processor services a maskable interrupt only if its requested interrupt level is greater than IL.	Read/Write	0	EIC interface only (3)
IH	IH is the interrupt handler mode bit. The processor sets IH to one when it takes an external interrupt.	Read/Write	0	EIC interface only (3)
EH (2)	EH is the exception handler mode bit. The processor sets EH to one when an exception occurs (including breaks). Software clears EH to zero when ready to handle exceptions again. EH is used by the MMU to determine whether a TLB miss exception is a fast TLB miss or a double TLB miss. In systems without an MMU, EH is always zero.	Read/Write	0	MMU only (3)
U (2)	U is the user mode bit. When U = 1, the processor operates in user mode. When U = 0, the processor operates in supervisor mode. In systems without an MMU, U is always zero.	Read/Write	0	MMU or MPU only (3)
PIE	PIE is the processor interrupt-enable bit. When PIE = 0, internal and maskable external interrupts and noninterrupt exceptions are ignored. When PIE = 1, internal and maskable external interrupts can be taken, depending on the status of the interrupt controller. Noninterrupt exceptions are unaffected by PIE.	Read/Write	0	Always

Other control registers (1)

- The **estatus register** holds a saved copy of the status register during nonbreak exception processing
- The **bstatus register** holds a saved copy of the status register during break exception processing
- The **ienable register** controls the handling of internal hardware interrupts
- The **ipending register** indicates the value of the interrupt signals driven into the processor

Other control registers (2)

- The **cpuid register** holds a constant value that is defined in the Nios II Processor parameter editor to uniquely identify each processor in a multiprocessor system
- When the extra exception information option is enabled, the Nios II processor provides information useful to system software for exception processing in the exception and badaddr registers when an exception occurs
- ...

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

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)

Nios II addressing modes.

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

EA = effective address

Value = a 16-bit signed number

X = a 16-bit signed displacement value

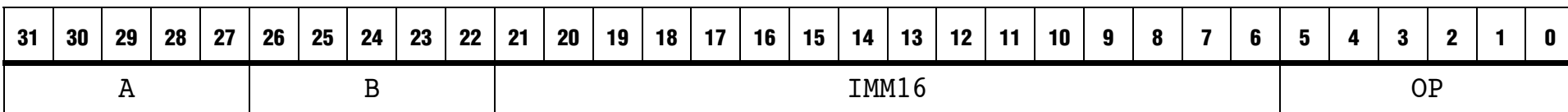
[*ri*] indicates the content of the register *ri*

Instruction formats (1)

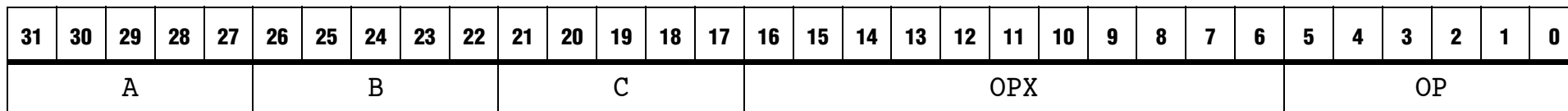
- RISC-style instructions (all 32-bit long)
 - Load/store architecture for data transfers
 - Arithmetic/logic instructions use registers
- Three instruction types:
 - I-type** OP dst_reg, src_reg, immediate
 - R-type** OP dst_reg, src_reg1, src_reg1
 - J-type** call label_or_address
- label_or_address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

Instruction formats (2)

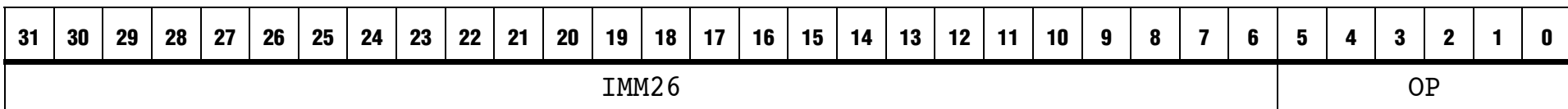
- **I-type instructions** include arithmetic and logical operations such as addi and andi; branch operations; load and store operations; and cache management operations.



- **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



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 register operands
- andi, ori, xori, nori have 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 immediate 16 bits left and clear lower 16 bits of immediate to zero

Move 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
                        or    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 offset

- Signed/unsigned comparison and branch:

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

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

- `beq`, `bne`, `bge`, `bgeu`, `bgt`, `bgtu`, `ble`, and `bleu`
- Unconditional branch beyond 16-bit offset:

```
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 = $PC_{31-28} : \text{Value26} : 00$
- Call with target in register: `callr ri`
- Return instruction: `ret`
 - Branches to address saved in r31 (ra)

Parameter Passing & Stack Frames

- Pass parameters in register or using stack
- Build stack frames for private work space and saving registers when nesting subroutine calls
- Called routine always saves frame ptr r28 (fp) before creating its own private work space
- Return addr r31 (ra) saved to enable nesting
- Use fp with displacement to access stack data:

```
ldw    r4, 8(fp)
```

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] < Val16
cmpltui ri, rj, Val16 // unsigned [rj] < 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 amount in rk`
`srl i ri , rj , Val5 //immediate shift amount`
- Shift right arithmetic `sra`, `srai`: same as above except that sign in bit rj_{31} is preserved
- Shift left logical `sl`, `sll`
- Rotate left `rol`, `rol i`
- Rotate right `ror` (no immediate version)

Control Instructions

- Special instructions to access control registers
- Read Control Register instruction:
`rdctl ri, ctlj // ri ← [ctlj]`
- Write Control Register instruction:
`wrctl ctlj, ri // ctlj ← [ri]`
- Instructions `trap`, `eret` deal with exceptions (similar to `call`, `ret` but for different purpose)
- Additional instructions for cache management

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` \Rightarrow `addi ri, rj, -Value16`
- Also can swap operands for comparisons:
`bgt ri, rj, LABEL` \Rightarrow `blt rj, ri, LABEL`
- Awareness of pseudoinstructions is not critical except when examining assembled code

Assembler Directives

- Nios II assembler directives conform to those defined by widely used GNU assembler:

<code>.org</code>	<i>Value</i>	(code/data origin)
<code>.equ</code>	<i>LABEL, Value</i>	(equate to label)
<code>.byte</code>	<i>expressions</i>	(define byte data)
<code>.hword</code>	<i>expressions</i>	(define halfwords)
<code>.word</code>	<i>expressions</i>	(define word data)
<code>.skip</code>	<i>Size</i>	(reserve bytes)
<code>.end</code>		(end of source 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: test if unsigned result < either operand:

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

- Carry bit is in r5

Carry/Overflow Detection for Add

- Overflow: 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

Input/Output

- Use I/O versions of Load/Store instructions
- Polling for program-controlled output:

```
movia r6, DATA_REG_ADDR
mov   r7, DATA_TO_SEND
movia r4, STATUS_REG_ADDR
```

L1:

```
ldbio r5, (r4)
andi  r5, r5, STATUS_FLAG_BIT
beq   r5, r0, L1
stbio r7, (r6)
```

Example Program

- Vector dot product performs multiplication and addition operations for array elements
 - Vectors A and B stored starting from address AVEC and BVEC, respectively
 - Vector size stored at address N
 - Result must be stored at address DOTPROD
 - Vector element, vector size and result are 32-bit wide


```

        movia    r2, AVEC          /* r2 points to vector A.          */
        movia    r3, BVEC          /* r3 points to vector B.          */
        movia    r4, N             /* Get the address N.              */
        ldw      r4, (r4)          /* r4 serves as a counter.         */
        mov      r5, r0            /* r5 accumulates the dot product. */
LOOP:   ldw      r6, (r2)          /* Get next element of vector A.   */
        ldw      r7, (r3)          /* Get next element of vector B.   */
        mul      r8, r6, r7        /* Compute the product of next pair. */
        add      r5, r5, r8        /* Add to previous sum.            */
        addi     r2, r2, 4          /* Increment pointer to vector A.  */
        addi     r3, r3, 4          /* Increment pointer to vector B.  */
        subi     r4, r4, 1          /* Decrement the counter.          */
        bgt      r4, r0, LOOP      /* Loop again if not done.         */
        movia    r2, DOTPROD        /* Store dot product                */
        stw      r5, (r2)          /* in memory.                       */

```

References

- Altera, “Nios II Processor Reference Handbook,” *n2cpu_nii5v1.pdf*
 - 2. Processor Architecture
 - 3. Programming Model/Exception Processing
 - 8. Instruction Set Reference