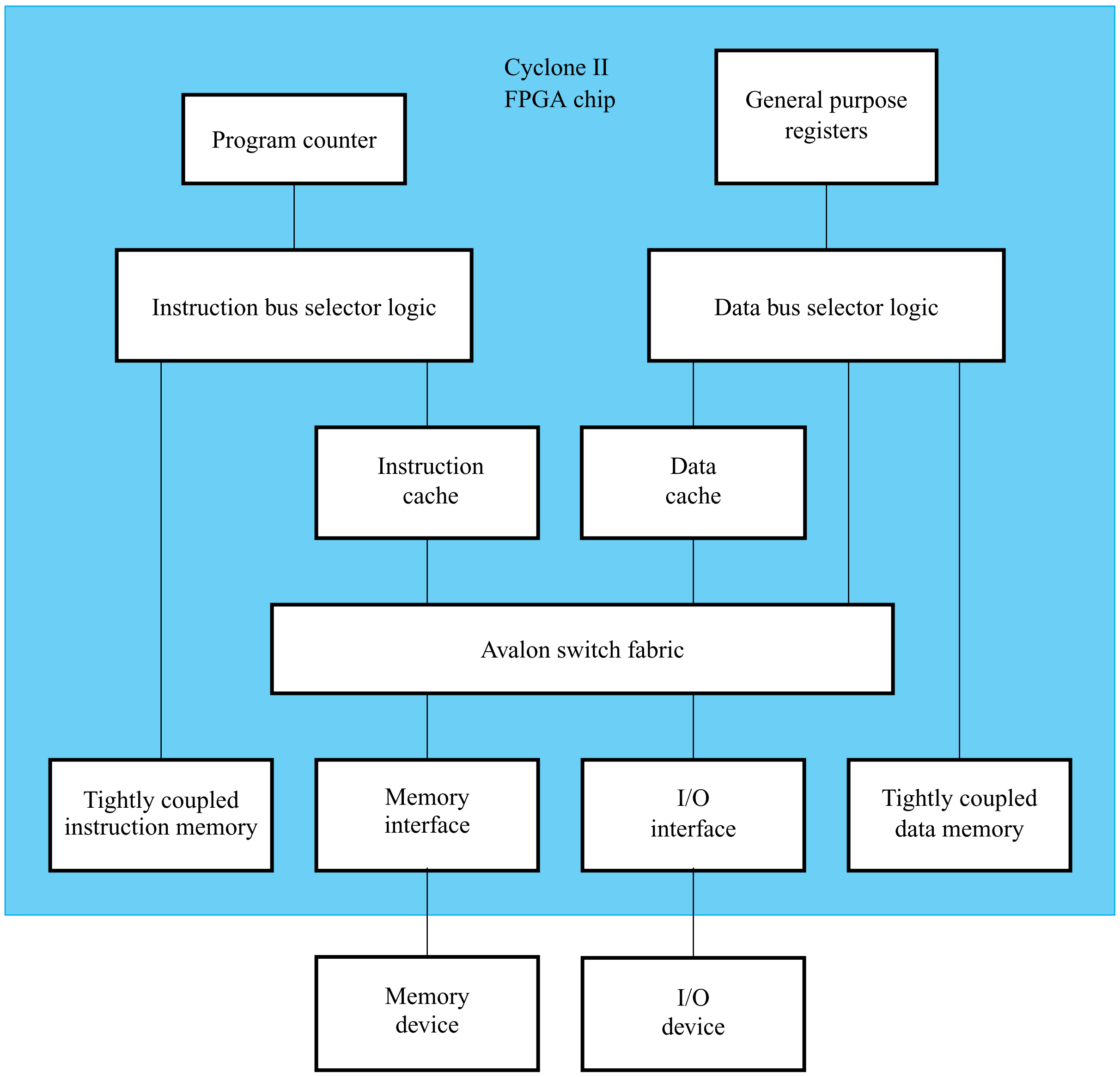


Figure 4. Memory and I/O organization.

5 Addressing

The Nios II processor issues 32-bit addresses. The memory space is byte-addressable. Instructions can read and write *words* (32 bits), *halfwords* (16 bits), or *bytes* (8 bits) of data. Reading or writing to an address that does not correspond to an existing memory or I/O location produces an undefined result.

There are five addressing modes provided:

- *Immediate mode* – a 16-bit operand is given explicitly in the instruction. This value may be sign extended to produce a 32-bit operand in instructions that perform arithmetic operations.
- *Register mode* – the operand is in a processor register
- *Displacement mode* – the effective address of the operand is the sum of the contents of a register and a signed 16-bit displacement value given in the instruction
- *Register indirect mode* – the effective address of the operand is the contents of a register specified in the instruction. This is equivalent to the displacement mode where the displacement value is equal to 0.

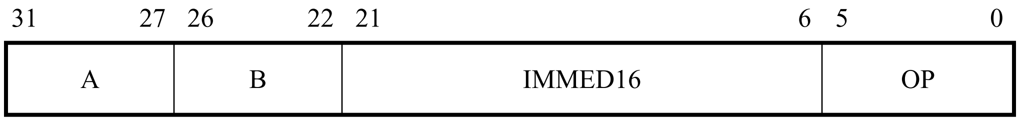
- *Absolute mode* – a 16-bit absolute address of an operand can be specified by using the displacement mode with register *r0* which always contains the value 0.

6 Instructions

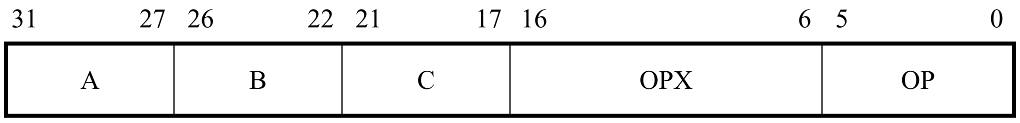
All Nios II instructions are 32-bits long. In addition to machine instructions that are executed directly by the processor, the Nios II instruction set includes a number of *pseudoinstructions* that can be used in assembly language programs. The Assembler replaces each pseudoinstruction by one or more machine instructions.

Figure 5 depicts the three possible instruction formats: I-type, R-type and J-type. In all cases the six bits b_{5-0} denote the OP code. The remaining bits are used to specify registers, immediate operands, or extended OP codes.

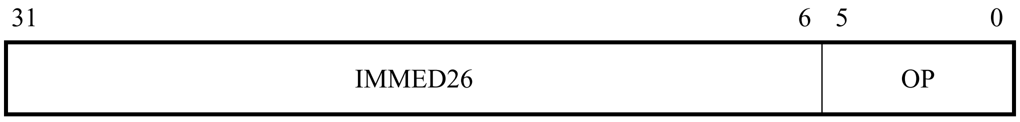
- I-type – Five-bit fields A and B are used to specify general purpose registers. A 16-bit field IMMED16 provides immediate data which can be sign extended to provide a 32-bit operand.
- R-type – Five-bit fields A, B and C are used to specify general purpose registers. An 11-bit field OPX is used to extend the OP code.
- J-type – A 26-bit field IMMED26 contains an unsigned immediate value. This format is used only in the Call instruction.



(a) I-type



(b) R-type



(c) J-type

Figure 5. Formats of Nios II instructions.

The following subsections discuss briefly the main features of the Nios II instruction set. For a complete description of the instruction set, including the details of how each instruction is encoded, the reader should consult the *Nios II Processor Reference Handbook*.

6.1 Load and Store Instructions

Load and Store instructions are used to move data between memory (and I/0 interfaces) and the general purpose registers. They are of I-type. For example, the Load Word instruction

```
ldw  rB, byte_offset(rA)
```

determines the effective address of a memory location as the sum of a byte_offset value and the contents of register A. The 16-bit byte_offset value is sign extended to 32 bits. The 32-bit memory operand is loaded into register B.

For instance, assume that the contents of register $r4$ are 1260_{10} and the byte_offset value is 80_{10}. Then, the instruction

```
ldw  r3, 80(r4)
```

loads the 32-bit operand at memory address 1340_{10} into register $r3$.

The Store Word instruction has the format

```
stw  rB, byte_offset(rA)
```

It stores the contents of register B into the memory location at the address computed as the sum of the byte_offset value and the contents of register A.

There are Load and Store instructions that use operands that are only 8 or 16 bits long. They are referred to as Load/Store Byte and Load/Store Halfword instructions, respectively. Such Load instructions are:

- ldb (Load Byte)
- ldbu (Load Byte Unsigned)
- ldh (Load Halfword)
- ldhu (Load Halfword Unsigned)

When a shorter operand is loaded into a 32-bit register, its value has to be adjusted to fit into the register. This is done by sign extending the 8- or 16-bit value to 32 bits in the ldb and ldh instructions. In the ldbu and ldhu instructions the operand is zero extended.

The corresponding Store instructions are:

- stb (Store Byte)
- sth (Store Halfword)

The stb instruction stores the low byte of register B into the memory byte specified by the effective address. The sth instruction stores the low halfword of register B. In this case the effective address must be halfword aligned.

Each Load and Store instruction has a version intended for accessing locations in I/O device interfaces. These instructions are:

- ldwio (Load Word I/O)
- ldbio (Load Byte I/O)
- ldbuio (Load Byte Unsigned I/O)
- ldhio (Load Halfword I/O)
- ldhuio (Load Halfword Unsigned I/O)
- stwio (Store Word I/O)
- stbio (Store Byte I/O)
- sthio (Store Halfword I/O)

The difference is that these instructions bypass the cache, if one exists.

6.2 Arithmetic Instructions

The arithmetic instructions operate on the data that is either in the general purpose registers or given as an immediate value in the instruction. These instructions are of R-type or I-type, respectively. They include:

- add (Add Registers)
- addi (Add Immediate)
- sub (Subtract Registers)
- subi (Subtract Immediate)
- mul (Multiply)
- muli (Multiply Immediate)
- div (Divide)
- divu (Divide Unsigned)

The Add instruction

```
add  rC, rA, rB
```

adds the contents of registers A and B, and places the sum into register C.

The Add Immediate instruction

```
addi  rB, rA, IMMED16
```

adds the contents of register A and the sign-extended 16-bit operand given in the instruction, and places the result into register B. The addition operation in these instructions is the same for both signed and unsigned operands; there are no condition flags that are set by the operation. This means that when unsigned operands are added, the carry from the most significant bit position has to be detected by executing a separate instruction. Similarly, when signed operands are added, the arithmetic overflow has to be detected separately. The detection of these conditions is dicussed in section 6.11.

The Subtract instruction

```
sub  rC, rA, rB
```

subtracts the contents of register B from register A, and places the result into register C. Again, the carry and overflow detection has to be done by using additional instructions, as explained in section 6.11.

The immediate version, subi, is a pseudoinstruction implemented as

```
addi  rB, rA, -IMMED16
```

The Multiply instruction

```
mul  rC, rA, rB
```

multiplies the contents of registers A and B, and places the low-order 32 bits of the product into register C. The operands are treated as unsigned numbers. The carry and overflow detection has to be done by using additional instructions. In the immediate version

```
muli  rB, rA, IMMED16
```

the 16-bit immediate operand is sign extended to 32 bits.

The Divide instruction

div rC, rA, rB

divides the contents of register A by the contents of register B and places the integer portion of the quotient into register C. The operands are treated as signed integers. The divu instruction is performed in the same way except that the operands are treated as unsigned integers.

6.3 Logic Instructions

The logic instructions provide the AND, OR, XOR, and NOR operations. They operate on data that is either in the general purpose registers or given as an immediate value in the instruction. These instructions are of R-type or I-type, respectively.

The AND instruction

and rC, rA, rB

performs a bitwise logical AND of the contents of registers A and B, and stores the result in register C. Similarly, the instructions or, xor and nor perform the OR, XOR and NOR operations, respectively.

The AND Immediate instruction

andi rB, rA, IMMED16

performs a bitwise logical AND of the contents of register A and the IMMED16 operand which is zero-extended to 32 bits, and stores the result in register B. Similarly, the instructions ori, xori and nori perform the OR, XOR and NOR operations, respectively.

It is also possible to use the 16-bit immediate operand as the 16 high-order bits in the logic operations, in which case the low-order 16 bits of the operand are zeros. This is accomplished with the instructions:

- andhi   (AND High Immediate)
- orhi   (OR High Immediate)
- xorhi   (XOR High Immediate)

6.4 Move Instructions

The Move instructions copy the contents of one register into another, or they place an immediate value into a register. They are pseudoinstructions implemented by using other instructions. The instruction

mov rC, rA

copies the contents of register A into register C. It is implemented as

add rC, rA, r0

The Move Immediate instruction

movi rB, IMMED16

sign extends the IMMED16 value to 32 bits and loads it into register B. It is implemented as

addi rB, r0, IMMED16

The Move Unsigned Immediate instruction

movui rB, IMMED16

zero extends the IMMED16 value to 32 bits and loads it into register B. It is implemented as

ori rB, r0, IMMED16

The Move Immediate Address instruction

movia rB, LABEL

loads a 32-bit value that corresponds to the address *LABEL* into register B. It is implemented as:

```
orhi    rB, r0, %hi(LABEL)
ori     rB, rB, %lo(LABEL)
```

The %hi(LABEL) and %lo(LABEL) are the Assembler macros which extract the high-order 16 bits and the low-order 16 bits, respectively, of a 32-bit value *LABEL*. The orhi instruction sets the high-order bits of register B, followed by the ori instruction which sets the low-order bits of B. Note that two instructions are used because the I-type format provides for only a 16-bit immediate operand.

6.5 Comparison Instructions

The Comparison instructions compare the contents of two registers or the contents of a register and an immediate value, and write either 1 (if true) or 0 (if false) into the result register. They are of R-type or I-type, respectively. These instructions correspond to the equality and relational operators in the C programming language.

The Compare Less Than Signed instruction

cmplt rC, rA, rB

performs the comparison of signed numbers in registers A and B, rA $<$ rB, and writes a 1 into register C if the result is true; otherwise, it writes a 0.

The Compare Less Than Unsigned instruction

cmpltu rC, rA, rB

performs the same function as the cmplt instruction, but it treats the operands as unsigned numbers.

Other instructions of this type are:

- cmpeq rC, rA, rB (Comparison rA == rB)
- cmpne rC, rA, rB (Comparison rA != rB)
- cmpge rC, rA, rB (Signed comparison rA $>$= rB)
- cmpgeu rC, rA, rB (Unsigned comparison rA $>$= rB)
- cmpgt rC, rA, rB (Signed comparison rA $>$ rB)
  This is a pseudoinstruction implemented as the cmplt instruction by swapping its rA and rB operands.
- cmpgtu rC, rA, rB (Unsigned comparison rA $>$ rB)
  This is a pseudoinstruction implemented as the cmpltu instruction by swapping its rA and rB operands.
- cmple rC, rA, rB (Signed comparison rA $<$= rB)
  This is a pseudoinstruction implemented as the cmpge instruction by swapping its rA and rB operands.
- cmpleu rC, rA, rB (Unsigned comparison rA $<$= rB)
  This is a pseudoinstruction implemented as the cmpgeu instruction by swapping its rA and rB operands.

The immediate versions of the Comparison instructions involve an immediate operand. For example, the Compare Less Than Signed Immediate instruction

cmplti rB, rA, IMMED16

compares the signed number in register A with the sign-extended immediate operand. It writes a 1 into register B if rA $<$ IMMED16; otherwise, it writes a 0.

The Compare Less Than Unsigned Immediate instruction

cmpltui rB, rA, IMMED16

compares the unsigned number in register A with the zero-extended immediate operand. It writes a 1 into register B if rA $<$ IMMED16; otherwise, it writes a 0.

Other instructions of this type are:

- cmpeqi rB, rA, IMMED16 (Comparison rA == IMMED16)
- cmpnei rB, rA, IMMED16 (Comparison rA != IMMED16)
- cmpgei rB, rA, IMMED16 (Signed comparison rA >= IMMED16)
- cmpgeui rB, rA, IMMED16 (Unsigned comparison rA >= IMMED16)
- cmpgti rB, rA, IMMED16 (Signed comparison rA > IMMED16)
  This is a pseudoinstruction which is implemented by using the cmpgei instruction with an immediate value IMMED16 + 1.
- cmpgtui rB, rA, IMMED16 (Unsigned comparison rA > IMMED16)
  This is a pseudoinstruction which is implemented by using the cmpgeui instruction with an immediate value IMMED16 + 1.
- cmplei rB, rA, IMMED16 (Signed comparison rA <= IMMED16)
  This is a pseudoinstruction which is implemented by using the cmplti instruction with an immediate value IMMED16 + 1.
- cmpleui rB, rA, IMMED16 (Unsigned comparison rA <= IMMED16)
  This is a pseudoinstruction which is implemented by using the cmpltui instruction with an immediate value IMMED16 + 1.

6.6 Shift Instructions

The Shift instructions shift the contents of a register either to the right or to the left. They are of R-type. They correspond to the shift operators, >> and <<, in the C programming language. These instructions are:

- srl rC, rA, rB (Shift Right Logical)
- srli rC, rA, IMMED5 (Shift Right Logical Immediate)
- sra rC, rA, rB (Shift Right Arithmetic)
- srai rC, rA, IMMED5 (Shift Right Arithmetic Immediate)
- sll rC, rA, rB (Shift Left Logical)
- slli rC, rA, IMMED5 (Shift Left Logical Immediate)

The srl instruction shifts the contents of register A to the right by the number of bit positions specified by the five least-significant bits (number in the range 0 to 31) in register B, and stores the result in register C. The vacated bits on the left side of the shifted operand are filled with 0s.

The srli instruction shifts the contents of register A to the right by the number of bit positions specified by the five-bit unsigned value, IMMED5, given in the instruction.

The sra and srai instructions perform the same actions as the srl and srli instructions, except that the sign bit, rA_{31}, is replicated into the vacated bits on the left side of the shifted operand.

The sll and slli instructions are similar to the srl and srli instructions, but they shift the operand in register A to the left and fill the vacated bits on the right side with 0s.

6.7 Rotate Instructions

There are three Rotate instructions, which use the R-type format:

- ror rC, rA, rB (Rotate Right)
- rol rC, rA, rB (Rotate Left)
- roli rC, rA, IMMED5 (Rotate Left Immediate)

The ror instruction rotates the bits of register A in the left-to-right direction by the number of bit positions specified by the five least-significant bits (number in the range 0 to 31) in register B, and stores the result in register C.

The rol instruction is similar to the ror instruction, but it rotates the operand in the right-to-left direction.

The roli instruction rotates the bits of register A in the right-to-left direction by the number of bit positions specified by the five-bit unsigned value, IMMED5, given in the instruction, and stores the result in register C.

6.8 Branch and Jump Instructions

The flow of execution of a program can be changed by executing Branch or Jump instructions. It may be changed either unconditionally or conditionally.

The Jump instruction

```
jmp rA
```

transfers execution unconditionally to the address contained in register A.

The Unconditional Branch instruction

```
br LABEL
```

transfers execution unconditionally to the instruction at address *LABEL*. This is an instruction of I-type, in which a 16-bit immediate value (interpreted as a signed number) specifies the offset to the branch target instruction. The offset is the distance in bytes from the instruction that immediately follows br to the address *LABEL*.

Conditional transfer of execution is achieved with the Conditional Branch instructions, which compare the contents of two registers and cause a branch if the result is true. These instructions are of I-type and the offset is determined as explained above for the br instruction.

The Branch if Less Than Signed instruction

blt rA, rB, LABEL

performs the comparison rA $<$ rB, treating the contents of the registers as signed numbers.

The Branch if Less Than Unsigned instruction

bltu rA, rB, LABEL

performs the comparison rA $<$ rB, treating the contents of the registers as unsigned numbers.

The other Conditional Branch instructions are:

- beq rA, rB, LABEL   (Comparison rA == rB)
- bne rA, rB, LABEL   (Comparison rA != rB)
- bge rA, rB, LABEL   (Signed comparison rA $>$= rB)
- bgeu rA, rB, LABEL   (Unsigned comparison rA $>$= rB)
- bgt rA, rB, LABEL   (Signed comparison rA $>$ rB)
  This is a pseudoinstruction implemented as the blt instruction by swapping the register operands.
- bgtu rA, rB, LABEL   (Unsigned comparison rA $>$ rB)
  This is a pseudoinstruction implemented as the bltu instruction by swapping the register operands.
- ble rA, rB, LABEL   (Signed comparison rA $<$= rB)
  This is a pseudoinstruction implemented as the bge instruction by swapping the register operands.
- bleu rA, rB, LABEL   (Unsigned comparison rA $<$= rB)
  This is a pseudoinstruction implemented as the bgeu instruction by swapping the register operands.

6.9 Subroutine Linkage Instructions

Nios II has two instructions for calling subroutines. The Call Subroutine instruction

call LABEL

is of J-type, which includes a 26-bit unsigned immediate value (IMMED26). The instruction saves the return address (which is the address of the next instruction) in register $r31$. Then, it transfers control to the instruction at address *LABEL*. This address is determined by concatenating the four high-order bits of the Program Counter with the IMMED26 value as follows

$$\text{Jump address} = \text{PC}_{31-28} : \text{IMMED26} : 00$$

Note that the two least-significant bits are 0 because Nios II instructions must be aligned on word boundaries.

The Call Subroutine in Register instruction

callr rA

is of R-type. It saves the return address in register $r31$ and then transfers control to the instruction at the address contained in register A.

Return from a subroutine is performed with the instruction

ret

This instruction transfers execution to the address contained in register $r31$.

6.10 Control Instructions

The Nios II control registers can be read and written by special instructions. The Read Control Register instruction

```
rdctl  rC, ctlN
```

copies the contents of control register *ctlN* into register C.

The Write Control Register instruction

```
wrctl  ctlN, rA
```

copies the contents of register A into the control register *ctlN*.

There are two instructions provided for dealing with exceptions: trap and eret. They are similar to the call and ret instructions, but they are used for exceptions. Their use is discussed in section 8.2.

The instructions break and bret generate breaks and return from breaks. They are used exclusively by the software debugging tools.

The Nios II cache memories are managed with the instructions: flushd (Flush Data Cache Line), flushi (Flush Instruction Cache Line), initd (Initialize Data Cache Line), and initi (Initialize Instruction Cache Line). These instructions are discussed in section 9.1.

6.11 Carry and Overflow Detection

As pointed out in section 6.2, the Add and Subtract instructions perform the corresponding operations in the same way for both signed and unsigned operands. The possible carry and arithmetic overflow conditions are not detected, because Nios II does not contain condition flags that might be set as a result. These conditions can be detected by using additional instructions.

Consider the Add instruction

```
add  rC, rA, rB
```

Having executed this instruction, a possible occurrence of a carry out of the most-significant bit (C_{31}) can be detected by checking whether the unsigned sum (in register C) is less than one of the unsigned operands. For example, if this instruction is followed by the instruction

```
cmpltu  rD, rC, rA
```

then the carry bit will be written into register D.

Similarly, if a branch is required when a carry occurs, this can be accomplished as follows:

```
add     rC, rA, rB
bltu    rC, rA, LABEL
```

A test for arithmetic overflow can be done by checking the signs of the summands and the resulting sum. An overflow occurs if two positive numbers produce a negative sum, or if two negative numbers produce a positive sum. Using this approach, the overflow condition can control a conditional branch as follows:

```
add    rC, rA, rB        /* The required Add operation */
xor    rD, rC, rA        /* Compare signs of sum and rA */
xor    rE, rC, rB        /* Compare signs of sum and rB */
and    rD, rD, rE        /* Set D31 = 1 if ((A31 == B31) != C31) */
blt    rD, r0, LABEL     /* Branch if overflow occurred */
```

A similar approach can be used to detect the carry and overflow conditions in Subtract operations. A carry out of the most-significant bit of the resulting difference can be detected by checking whether the first operand is less than the second operand. Thus, the carry can be used to control a conditional branch as follows:

```
sub    rC, rA, rB
bltu   rA, rB, LABEL
```

The arithmetic overflow in a Subtract operation is detected by comparing the sign of the generated difference with the signs of the operands. Overflow occurs if the operands in registers A and B have different signs, and the sign of the difference in register C is different than the sign of A. Thus, a conditional branch based on the arithmetic overflow can be achieved as follows:

```
sub    rC, rA, rB        /* The required Subtract operation */
xor    rD, rA, rB        /* Compare signs of rA and rB */
xor    rE, rA, rC        /* Compare signs of rA and rC */
and    rD, rD, rE        /* Set D31 = 1 if ((A31 != B31) && (A31 != C31)) */
blt    rD, r0, LABEL     /* Branch if overflow occurred */
```

7 Assembler Directives

The Nios II Assembler conforms to the widely used GNU Assembler, which is software available in the public domain. Thus, the GNU Assembler directives can be used in Nios II programs. Assembler directives begin with a period. We describe some of the more frequently used assembler directives below.

.ascii "*string*"...

A string of ASCII characters is loaded into consecutive byte addresses in the memory. Multiple strings, separated by commas, can be specified.

.asciz "*string*"...

This directive is the same as .ascii, except that each string is followed (terminated) by a zero byte.

.byte *expressions*

Expressions separated by commas are specified. Each expression is assembled into the next byte. Examples of expressions are: 8, 5 + LABEL, and K − 6.

.data

Identifies the data that should be placed in the data section of the memory. The desired memory location for the data section can be specified in the Altera Debug Client's system configuration window.

.end

Marks the end of the source code file; everything after this directive is ignored by the assembler.

.equ *symbol, expression*

Sets the value of *symbol* to *expression*.

.global *symbol*

Makes *symbol* visible outside the assembled object file.

.hword *expressions*

Expressions separated by commas are specified. Each expression is assembled into a 16-bit number.

.include "*filename*"

Provides a mechanism for including supporting files in a source program.

.org *new-lc*

Advances the location counter to *new-lc*. The .org directive may only increase the location counter, or leave it unchanged; it cannot move the location counter backwards.

.skip *size*

Emits the number of bytes specified in *size*; the value of each byte is zero.

.text

Identifies the code that should be placed in the text section of the memory. The desired memory location for the text section can be specified in the Altera Debug Client's system configuration window.

.word *expressions*

Expressions separated by commas are specified. Each expression is assembled into a 32-bit number.

8 Example Program

As an illustration of Nios II instructions and assembler directives, Figure 6 gives an assembly language program that computes a dot product of two vectors, *A* and *B*. The vectors have n elements. The required computation is

$$\text{Dot product} = \sum_{i=0}^{n-1} \text{A}(i) \times \text{B}(i)$$

The vectors are stored in memory locations at addresses *AVECTOR* and *BVECTOR*, respectively. The number of elements, n, is stored in memory location N. The computed result is written into memory location *DOT_PRODUCT*. Each vector element is assumed to be a signed 32-bit number.

```
.include  "nios_macros.s"
.global   _start
_start:
          movia  r2, AVECTOR              /* Register r2 is a pointer to vector A */
          movia  r3, BVECTOR              /* Register r3 is a pointer to vector B */
          movia  r4, N
          ldw    r4, 0(r4)                /* Register r4 is used as the counter for loop iterations */
          add    r5, r0, r0               /* Register r5 is used to accumulate the product */
LOOP:     ldw    r6, 0(r2)                /* Load the next element of vector A */
          ldw    r7, 0(r3)                /* Load the next element of vector B */
          mul    r8, r6, r7               /* Compute the product of next pair of elements */
          add    r5, r5, r8               /* Add to the sum */
          addi   r2, r2, 4                /* Increment the pointer to vector A */
          addi   r3, r3, 4                /* Increment the pointer to vector B */
          subi   r4, r4, 1                /* Decrement the counter */
          bgt    r4, r0, LOOP             /* Loop again if not finished */
          stw    r5, DOT_PRODUCT(r0)      /* Store the result in memory */
STOP:     br     STOP

N:
.word     6                               /* Specify the number of elements */
AVECTOR:
.word     5, 3, −6, 19, 8, 12             /* Specify the elements of vector A */
BVECTOR:
.word     2, 14, −3, 2, −5, 36            /* Specify the elements of vector B */
DOT_PRODUCT:
.skip     4
```

Figure 6. A program that computes the dot product of two vectors.

Note that the program ends by continuously looping on the last Branch instruction. If instead we wanted to pass control to debugging software, we could replace this br instruction with the break instruction.

The program includes the assembler directive

```
.include  "nios_macros.s"
```

which informs the Assembler to use some macro commands that have been created for the Nios II processor. In this program, the macro used converts the movia pseudoinstruction into two OR instructions as explained in section 6.4.

The directive

```
.global  _start
```

indicates to the Assembler that the label *_start* is accessible outside the assembled object file. This label is the default label we use to indicate to the Linker program the beginning of the application program.

The program includes some sample data. It illustrates how the .word directive can be used to load data items into memory. The memory locations involved are those that follow the location occupied by the br instruction. Since we have not explicitly specified the starting address of the program itself, the assembled code will be loaded in memory starting at address 0.

To execute the program in Figure 6 on Altera's DE2 board, it is necessary to implement a Nios II processor and its memory (which can be just the on-chip memory of the Cyclone II FPGA). Since the program includes the