NIOS II Assembly

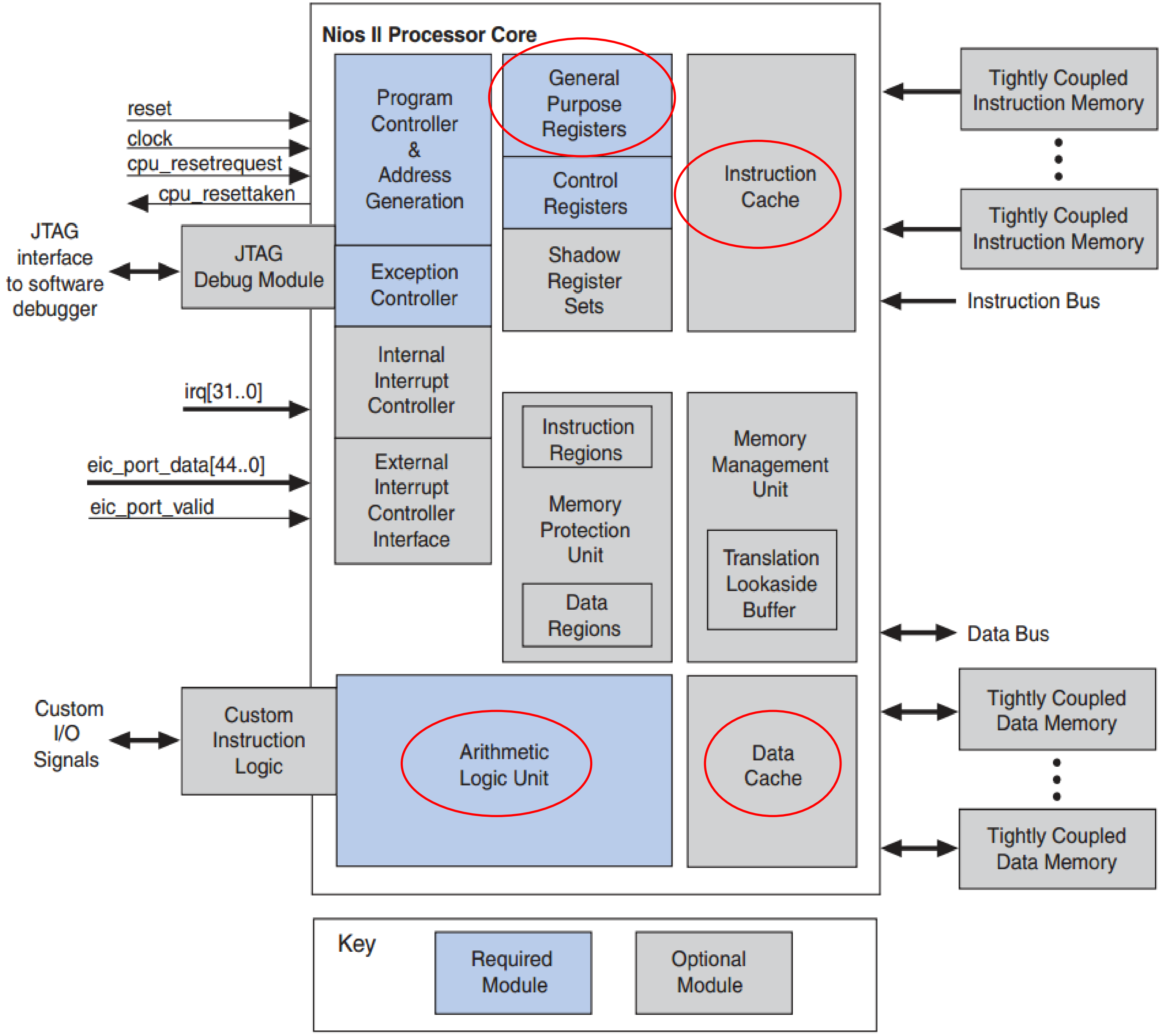
UNIVERSITY OF WASHINGTON
ELECTRICAL ENGINEERING

What is Assembly?

- Assembly is a low level coding language specific to your processor's architecture
- You manipulate data located in registers using very basic instructions
- Any lower level would involve coding in 1s and 0s
- Programs written in high level languages like Java, C++, etc goes through a compiler that turns it into assembly
- That assembly then goes through an assembler and linker to the machine language (1s and 0s) for your specific platform

NIOS II Processor Architecture

Figure 2–1. Nios II Processor Core Block Diagram



NIOS II Instruction Set

- Similar to assembly for most other architectures
- Conforms to GNU Assembler
- Refer to Altera documentation for full details
  - http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf
  - http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf
- Useful introductions & the examples can be found:
  - ftp://ftp.altera.com/up/pub/Tutorials/DE2/Computer_Organization/tut_nios2_introduction.pdf

Registers

- NIOS II has 32 general purpose registers (gpr)
- You will use these for most of your assembly code
- Refer to documentation for usage: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

Table 3–5. The Nios II General-Purpose Registers

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

Notes:

(1) r25 is used exclusively by the JTAG debug module. It is used as the breakpoint temporary (bt) register in the normal register set. In shadow register sets, r25 is reserved.

(2) r30 is used as the breakpoint return address (ba) in the normal register set, and as the shadow register set status (sstatus) in each shadow register set. For details about sstatus, refer to *The Status Register* section.

Arithmetic Logic Unit

- Responsible for arithmetic operations on CPU
- Includes arithmetic and logical operations (ALU), relational comparisons, and bit manipulations

Table 2–1. Operations Supported by the Nios II ALU

Category	Details
Arithmetic	The ALU supports addition, subtraction, multiplication, and division on signed and unsigned operands.
Relational	The ALU supports the equal, not-equal, greater-than-or-equal, and less-than relational operations (==, != >=, <) on signed and unsigned operands.
Logical	The ALU supports AND, OR, NOR, and XOR logical operations.
Shift and Rotate	The ALU supports shift and rotate operations, and can shift/rotate data by 0 to 31 bit positions per instruction. The ALU supports arithmetic shift right and logical shift right/left. The ALU supports rotate left/right.

Assembler Directives

- .global – symbol becomes visible outside assembled object file
- .include – includes supporting files to source
- .text – code (program) placed in text section of memory
- .data – data that is placed in data section of memory
- .word – 32bit expressions separated by commas
- .hword – 16bit expressions separated by commas
- .ascii – string of ASCII characters; can be comma separated strings
- .asciz – same as .ascii with a zero byte termination at each string end
- .skip – essentially allocates a set of memory for a variable
- .end – end of the source code file

Instruction Word Format

- 3 Types of instruction formats
  - I type, R type, J type
- 5 bit register fields (A, B, C)
- 6 bit opcode field (OP)
- 16 bit immediate data field (IMM16)
  - Meaning a 16 bit value, not register address

Table 8–1. I-Type Instruction Format

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
A					B					IMM16																OP					

Table 8–2. R-Type Instruction Format

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
A					B					C					OPX											OP					

Table 8–3. J-Type Instruction Format

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
IMM26																										OP					

NIOS II Assembly Programming

Format of Syntax (e.g. R type)

Label: **Opcode Operand, Operand, Operand** **#Comments**

Example

```
loop:                  # Label “loop”
    ldh r6, 100(r5)  # Load 16bit data at address in r5 + 100 bytes into r6
    ldh r7, 104(r5)  # Load 16bit data at address in  r5 + 104 bytes into r7
    add r8, r7, r6       # Add values in r6 and r7 into r8
    br loop              # Branch back to label called “loop”
```

Fields

- Label
  - A symbolic address with a location in the program. A label is optional but is useful if you wish to jump to a specific location within the program’s execution such as in loops or if statements
- Opcode
  - The instruction, macro, etc.
- Operand(s)
  - Can be constants (IMM16) or registers, etc
- Comments
  - Useful to comment on your code but this is optional. Comment should come after a “#” or “;” to indicate the end of the assembly instruction word

Multiply (mul)

Example:

mul r6, r7, r8

If r7 = 10 and r8 = 10, then r6 would get 100.

mul **multiply**

Operation:	$rC \leftarrow (rA \times rB)_{31..0}$
Assembler Syntax:	`mul rC, rA, rB`
Example:	`mul r6, r7, r8`
Description:	Multiplies rA times rB and stores the 32 low-order bits of the product to rC. The result is the same whether the operands are treated as signed or unsigned integers. Nios II processors that do not implement the `mul` instruction cause an unimplemented instruction exception.
Usage:	Carry Detection (unsigned operands): Before or after the multiply operation, the carry out of the MSB of rC can be detected using the following instruction sequence:

Carry Detection (unsigned operands)

- mul rC, rA, rB
- mulxuu rD, rA, rB   # rD is nonzero if carry occurred
- cmpne rD, rD, r0   # rD is 1 if carry occurred, 0 if not

- The mulxuu instruction writes a nonzero value into rD if the multiplication of unsigned numbers generates a carry (unsigned overflow). If a 0/1 carry detection result is desired, follow the mulxuu with the cmpne instruction.

Add (add)

Example:

add r6, r7, r8

If r7 = 10 and r8 = 10, then r6 would get 20

add **add**

Operation:	rC ← rA + rB
Assembler Syntax:	`add rC, rA, rB`
Example:	`add r6, r7, r8`
Description:	Calculates the sum of rA and rB. Stores the result in rC. Used for both signed and unsigned addition.
Usage:	Carry Detection (unsigned operands):

Following an `add` operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. The following code shows both cases:

Add Immediate (addi)

Example:

Addi rB, rA, -100

If rA = 103, then rB = rA -100 = 3

addi **add immediate**

Operation:	$rB \leftarrow rA + \sigma(\text{IMM16})$
Assembler Syntax:	`addi rB, rA, IMM16`
Example:	`addi r6, r7, -100`
Description:	Sign-extends the 16-bit immediate value and adds it to the value of rA. Stores the sum in rB.
Usage:	Carry Detection (unsigned operands):
	Following an `addi` operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. The following code shows both cases:

Branching and Jumping

Example (different conditions, e.g., greater than, bgt or bgtu):

bgt r6, r7, top_of_loop

If r6 = 10 and r7 = 0, then the program jumps to the instruction labeled by “top_of_loop:”

bgt **branch if greater than signed**

Operation:	if ((signed) rA > (signed) rB) then PC ← label else PC ← PC + 4
Assembler Syntax:	`bgt rA, rB, label`
Example:	`bgt r6, r7, top_of_loop`
Description:	If (signed) rA > (signed) rB, then `bgt` transfers program control to the instruction at label.
Pseudo-instruction:	`bgt` is implemented with the `blt` instruction by swapping the register operands.

Conditional Branching (Jumping)

Instruction	Description
bge bgeu bgt bgtu ble bleu blt bltu beq bne	These instructions provide relative branches that compare two register values and branch if the expression is true. Refer to the "Comparison Instructions" section of this chapter for a description of the relational operations implemented.

Unconditional Jumping

Table 3-47: Unconditional Jump and Call Instructions

Instruction	Description
`call`	This instruction calls a subroutine using an immediate value as the subroutine's absolute address, and stores the return address in register `ra`.
`callr`	This instruction calls a subroutine at the absolute address contained in a register, and stores the return address in register `ra`. This instruction serves the roll of dereferencing a C function pointer.
`ret`	The `ret` instruction is used to return from subroutines called by `call` or `callr`. `ret` loads and executes the instruction specified by the address in register `ra`.
`jmp`	The `jmp` instruction jumps to an absolute address contained in a register. `jmp` is used to implement switch statements of the C programming language.
`jmpi`	The `jmpi` instruction jumps to an absolute address using an immediate value to determine the absolute address.
`br`	This instruction branches relative to the current instruction. A signed immediate value gives the offset of the next instruction to execute.

Load Instructions

- Used to move data from memory & I/O to GPR
  - ldb – Load Byte (8 bit)
  - ldbu – Load Byte Unsigned (8 bit)
  - ldh – Load Halfword (16 bit)
  - ldhu – Load Halfword Unsigned (16 bit)
  - ldw – Load Word (32 bit)
- This processor is byte addressable so data in the memory can be "indirectly" accessed via the appropriate address (pre-loaded in the registers)
- Example: 0xF230

Load Word (ldw)

Example:

ldw r6, 100(r5)

Adds 100 bytes to r5 (moves pointer down 25 words, i.e., 100 bytes)

Loads the 32bit value at that location to r6

ldw / ldwio **load 32-bit word from memory or I/O peripheral**

Operation:	rB ← Mem32[rA + σ (IMM14)]
Assembler Syntax:	`ldw rB, byte_offset(rA)` `ldwio rB, byte_offset(rA)`
Example:	`ldw r6, 100(r5)`
Description:	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory word located at the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.
Usage:	In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldwio` acts like `ldw`. For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Store Word (stw)

Example:

stw r6, 100(r5)

If r5 = the address to an array, r6 would get the 25th element of that array

100 is the number of bytes, there are 4 bytes (32bits) per word

stw / stwio **store word to memory or I/O peripheral**

Operation:	Mem32[rA + σ (IMM16)] ← rB
Assembler Syntax:	`stw rB, byte_offset(rA)`
	`stwio rB, byte_offset(rA)`
Example:	`stw r6, 100(r5)`
Description:	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores rB to the memory location specified by the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.
Usage:	In processors with a data cache, this instruction may not generate an Avalon-MM data transfer immediately. Use the `stwio` instruction for peripheral I/O. In processors with a data cache, `stwio` bypasses the cache and is guaranteed to generate an Avalon-MM bus cycle. In processors without a data cache, `stwio` acts like `stw`.

Move Immediate Address (movia)

Example:

A pseudo-instruction that combines two others (include nios_macros.s)

movia r6, function_address

If the address of function_address is 0x3000, then r6 = 0x3000

movia **move immediate address into word**

Operation:	rB ← label
Assembler Syntax:	`movia rB, label`
Example:	`movia r6, function_address`
Description:	Writes the address of label to rB.
Pseudo-instruction:	`movia` is implemented as:
	`orhi rB, r0, %hiadj(label)`
	`addi rB, rB, %lo(label)`

Implementation of Sum of Products (SoP)

- It has been shown that SoP or Multiply Accumulate (MAC) is the key element for most DSP algorithms
- Let's go over some basic instructions and write assembly code to implement a sum of products
- Full assembly instruction set can be found in reference manual

$$Y = \sum_{n=0}^{N-1} a_n * x_n$$

$$Y = a_0 * x_0 + a_1 * x_1 + \cdots + a_{N-1} x_{N-1}$$

- Two basic operations:
    1. Multiplication
    2. Addition
- Two basic assembly instructions

SoP Code

- Code from the NIOS II tutorial
- Implements dot product

$$Y = \sum_{n=0}^{N-1} a_n * b_n$$

$$Y = a_0 * b_0 + a_1 * b_1 + \cdots + a_{N-1} b_{N-1}$$

- Essentially Sum of Products (SoP)

```
.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                 ← Only assign the address of variable 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.

Start

- Includes macros used by NIOS II to implement the pseudo-instruction “movia”
- _start is the default to indicate the start of the application program

```
.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.

Data

- Number of Samples, N = 6
- Size of DOT_PRODUCT = 32bits (_skip 4)

n	a_n	b_n
0	5	2
1	3	14
2	-6	-3
3	19	2
4	8	-5
5	12	36

```
.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.

Setup

- Puts addresses that point to the sample data (a_n and b_n) into registers r2 and r3
- Puts size of sample, N into r4
- Sets accumulator r5 to 0 (r0 is zero)

```
.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.

Loop

- Loads elements of vectors into registers r6 and r7
- Multiplies r6 (a) and r7 (x) and puts product into r8
- Update SoP by adding the product to r5
- Updates r2 and r3 pointers by adding 4 bytes to pointer (32 bit word)
- Decrements the counter N by 1

```
.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.

Store Data

- Compares r4 and r0
- If r4 > r0, then branch to label "LOOP"
- When the counter r4 decrements to 0, then the program won't branch anymore
- Store the result in r5 into the memory allocated to DOT_PRODUCT at that address location + zero
- End application

```
.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.

Calling Assembly Function in C program

- Write your assembly code in a .s file
- Use the ".global" directive at the top to allow this function to be accessed globally by your C program
- The label, sop: in this case, is the name of the function

```
.global sop
/*
r4 - data (x)
r5 - coefficients (h)
r6 - N
*/
sop:
        add r2, r0, r0
        LOOP:
                ldw r9, 0(r4)
                ldw r10, 0(r5)
                mul r8, r9, r10
                add r2, r2, r8
                addi r4, r4, 4
                addi r5, r5, 4
                subi r6, r6, 1
                bgt r6, r0, LOOP
        ret
```

Calling Assembly Function in C program

- Check the user guide to see the registers used for arguments and returns
- For this, r4, r5, r6 are the arguments
- r2 is the return value
- So in your C function you would call:
- int result = sop(x, h, N);

```
.global sop
/*
r4 - data (x)
r5 - coefficients (h)
r6 - N
*/
sop:
        add r2, r0, r0
        LOOP:
                ldw r9, 0(r4)
                ldw r10, 0(r5)
                mul r8, r9, r10
                add r2, r2, r8
                addi r4, r4, 4
                addi r5, r5, 4
                subi r6, r6, 1
                bgt r6, r0, LOOP
        ret
```

Calling Assembly Function in C program

C code

```c
#include <stdio.h>

main()
{
 int N=6;
 int x[6]={0,1,2,3,4,5};
 int h[6]={1,2,3,4,5,6};
 int result;

 result = sop(x, h, N);
 printf(“SoP = %d", result);
}
```

Assembly code

```
.global sop
/*
r4 - data (x), r5 - coefficients (h), r6 - N
*/
sop:
        add r2, r0, r0
        LOOP:
                ldw r9, 0(r4)
                ldw r10, 0(r5)
                mul r8, r9, r10
                add r2, r2, r8
                addi r4, r4, 4
                addi r5, r5, 4
                subi r6, r6, 1
                bgt r6, r0, LOOP
        ret
```

Inputs

- One input: always using r4
- Many inputs: using these register in order r4,r5,r6… (or use r4 as the pointer to memory)

Outputs

- One output: always using r2
- Many outputs: use r2 as the pointer to memory