IS1200: Suggested Solutions For Exercise CE 1
Nios II Assembly Instructions

January 22, 2015

1. Nios II Computer Architecture

(a) Table 3-1 in the Nios II Processor Reference Handbook lists processor registers. The table is reproduced in Figure 1.

Table 3–1. The Nios II General Purpose Registers

Register	Name	Function	Register	Name	Function
r0	zero	0x00000000	r16		
r1	at	Assembler Temporary	r17		
r2		Return Value	r18		
r3		Return Value	r19		
r4		Register Arguments	r20		
r5		Register Arguments	r21		
r6		Register Arguments	r22		
r7		Register Arguments	r23		
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 *(1)*
r15		Caller-Saved Register	r31	ra	Return Address

Notes to Table 3–1:
(1) This register is used exclusively by the JTAG debug module.

Figure 1: General purpose registers in Nios II (from Nios II Processor Reference Handbook).

- Register `r0` always contains zero.
- Register `r1` should not be used - the compiler uses it for some optimizations.
- Registers `r2-r7` and `r31` is used for function calls (also called method call or subroutine). This is dealt in a later exercise.
- Registers `r8-r15` can be used freely.

- Registers `r16-r23` can be used freely, provided you save them before use and restore them when done. This applies to the context of subroutines, which you will learn in a later exercise.
- Registers `r24` and `r29` should not be used except in case of interruption (interrupts) which is dealt in a later exercise.
- Registers `r25` and `r30` should not be used, they are used by the debugger.
- Registers `r26` and `r28` should not be modified, they are used by the compiler for reading and writing of local and global variables.
- Register `r27` is the stack pointer and is handled in a later exercise.

(b) In the Nios II processor, an instruction is always 32 bits (4 bytes) long.

(c) The program counter contains the address of the next instruction to be fetched and executed.

(d) When an instruction is fetched, the contents of the program counter is increased by 4, so that its value indicates the address of the next instruction that is te be fetched. *Important:* It is not enough to increase the program counter value by 1, because each instruction is 4 bytes long.

2. **Different basic types of assembler instructions.**

(a) Instructions to copy the data from memory to registers or conversely: the load (`ld`) and store (`st`) instruction in different varieties.

(b) Instructions to copy data from one register to another: there is only one instruction – `mov`.

(c) Instructions for making an arithmetic operation on two values: `add, sub, mul and div` in different varieties.

(d) Instructions for logical calculations: `and, or, xor and nor` in different varieties.

(e) Instructions to shift or rotate the bit pattern in a register: `sll, sra, srl, rol and ror` in different varieties.

(f) Instructions for comparing two values: `cmp` in different varieties.

(g) Instructions for jumping, i.e., changing the program counter: `jmp` and `br`.

Important: The instruction `jmpi` is not supported in the version of the Nios II processor used in lab exercises.

(h) Conditional branch instructions only change the program counter if the value in one or two registers meets a certain condition. These instructions are called `b*`, where `*` identifies the branch condition.

3. **Load and store instructions in Nios II.**

   (a) Addressing mode for load and store instructions in Nios II is always as follows. The contents of a register and a constant value contained in a portion of the instruction is added. The constant value can be both positive and negative. The result of the addition is used as the memory address. This addressing mode is usually called *indexed* or *base-relative* addressing.

   (b) Operand size: the most common size is 4 bytes (called a *word* in Nios II) - `ldw` and `stw`. Other possibilities are 2 bytes (a *half-word*) - `ldh` and `sth` - and 1 byte, `ldb` and `stb`.

   (c) Copying 4 bytes of memory to a register is shown in Figure 2. In Nios II, the bytes that are closest to address 0 in memory are copied to the least significant part of the register.

   When copying the other way i.e., from register to memory, the byte in the least significant part of the register is copied closest to address 0 in memory, so that the store operation is symmetric with Load operation. See Figure 3.

   (d) If 1 byte of memory is copied to a register, then it is placed in the least significant part of the register. All the other bits in the register are also changed. When the instruction `ldbu` is used, all the other bits in the register are set to zero. If instead, the instruction `ldb`, the other bits are set to the same value as the most significant bit of the copied byte. This process is called a *sign extension*. See Figure 4.

   (e) If 1 byte is copied from a register to memory with the instruction `stb`, the other bytes in the register are ignored. Only the least significant byte of the register is copied to the memory. Only one byte is written to memory.

4. **Addition and subtraction operations in Nios II.**

   (a) Addressing modes: `add, sub, mul, div` and `divu` all have their operands in registers. For `addi, subi` and `muli`, there is a constant operand that is part of the instruction.

   (b) Only possible operand size is 4 bytes (32 bits).

   (c) Possible number representations are integers with and without the sign bit. Without the sign bit, numbers range from 0 to $2^{32}-1$; with the sign bit, numbers from - (2^{31}) to + ($2^{31}-1$). The instructions are defined so that the difference between signed and unsigned numbers only affects the division operation, which is available in two variants – `div` and `divu`.

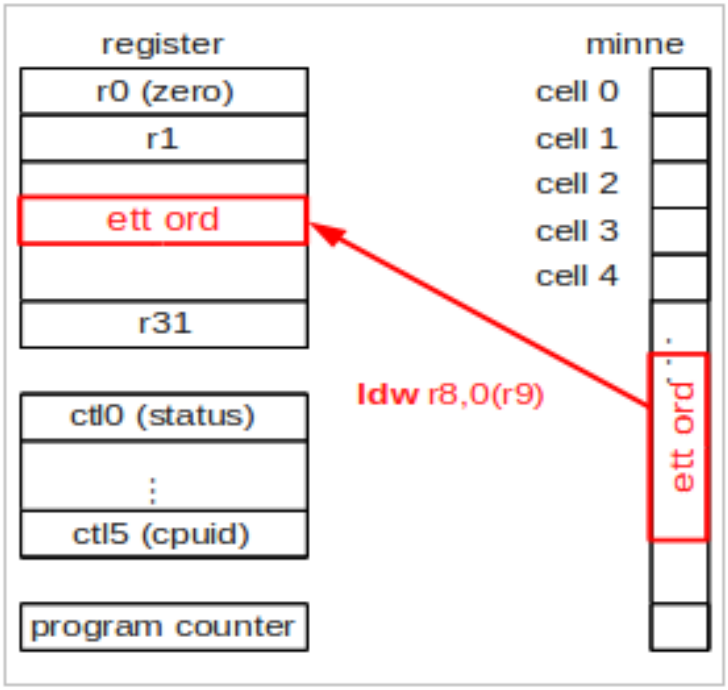

Figure 2: Copying from memory to register

5. **A first assembly language program - straight code with load and store instructions.**

   Program declarations

```
int a;
int b;
int c;
```

   translate to

```
.data        # indicates that variable declarations come after .data
.align 2     # needed to place a word on a valid address
a: .word 0   # reserves space for a word (4 bytes) with content 0
b: .word 0
c: .word 0
```

   Program line `c = a + b;` then translates to

```
.text             # indicates that code comes after the .text
.align 2          # needed for an instruction to end up on a valid address
movia r8, a       # move the address of variable a in r8
ldw r9,0 (r8)     # read the value of variable a from memory to r9
```

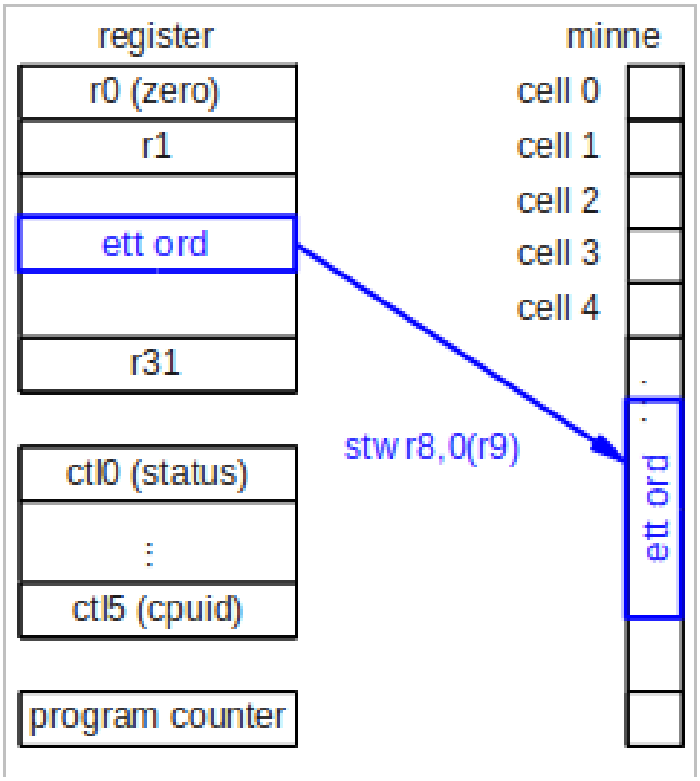

Figure 3: Copying from register to memory

```
5   movia r8, b       # move the address of the variable b to r8
6   ldw r10,0 (r8)    # read the value of variable b from memory to r10
7   add r11, r10, r9  # adding the values of variables a and b
8   movia r8, c       # move the address of variable c in r8
9   stw r11,0 (r8)    # write the sum of variable c to memory
```

Note: In the Nios II assembly language, the # character says the rest of the line is a comment.

6. **Comparison instructions in Nios II.**

   The comparison instruction is `cmp` (stands for compare). Its variants are: `cmpeq, cmpne, cmpge, cmpgeu, cmpgt, cmpgtu, cmple, cmpleu, cmplt, cmpltu cmpeqi, cmpnei, cmpgei, cmpgeui, cmpgti, cmpgtui, cmplei, cmpleui, cmplti, cmpltui`

   The instruction `cmp` sets the destination register to 1 if the condition is met, and to 0 if the condition is not met. The type of condition is encoded in the name of the `cmp` instruction (letters * in *cmp**).

   The result of the comparison is stored as true (1) or false (0) in the destination register. The contents of the register can be used as criteria

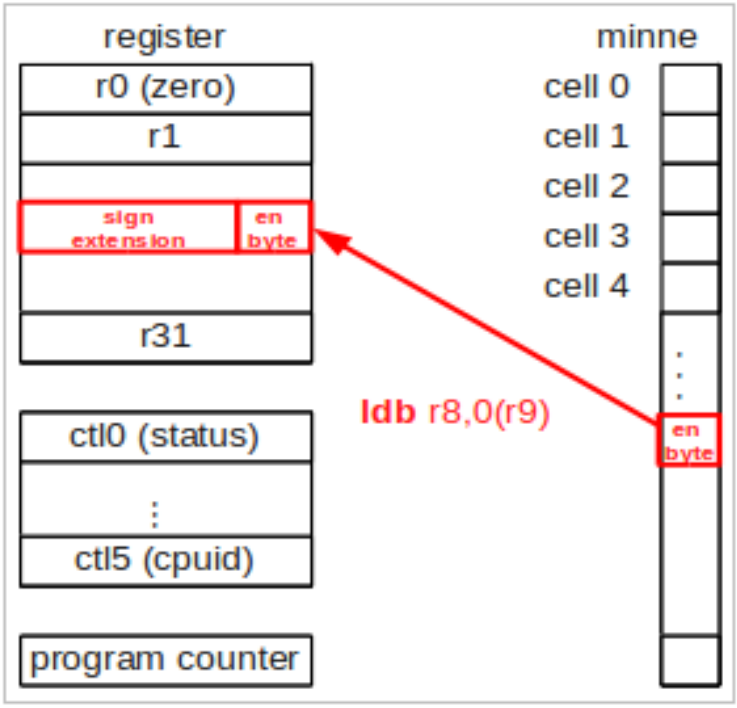

Figure 4: Sign extension

in a subsequent conditional branch instruction, or as a constant operand to other `cmp` instructions. The latter may be needed to calculate the complicated conditions.

One rule to remember the right `cmp` instruction in Nios II assembly is to put the letter combination for the condition between the two source operands and then read out the result. Example: Think of `cmpge r1, r2, r3` as `cmp r1, r2 ge r3`; then read out the condition as *r2 greater or equal to r3*.

Condition encoding: `eq` - equal to, `ne` - not equal to, `ge` - greater than or equal to, `gt` - greater than, `le` - less than or equal, `lt` - less than.

Compare instructions are available with and without *u* in their encoding. The `u` stands for *unsigned*, which means that the most significant bit is interpreted as a normal bit and not as a sign bit. An example: `cmpgtu` interprets 1111 1111 1111 1111 1111 1111 1111 1111 which +4 billion, which is larger than 0000 0000 0000 0000 0000 0000 0000 0001 (which of course is the number +1).

The instruction `cmpgt` without `u` interprets 1111 1111 1111 1111 1111 1111 1111 1111 as -1 which is less than +1 (which of course is still being written as 0000 0000 0000 0000 0000 0000 0000 0001).

Tips! Think about it and answer: why don't the terms `eq` and `ne` exist in unsigned versions?

Each `cmp` instruction is also available in versions with and without `i` in their encoding. The `i` stands as usual for *immediate* and states that one operand is a constant contained in the instruction's binary code. The constant in the immediate field is indicated using 16 bits. When comparing unsigned values, the constant is extended with zeros from 16 to 32 bits. When comparing signed values (without `u`) the constant is sign extended from 16 to 32 bits. The effect of these measures is that the resulting 32 bit code always corresponds to the same value as the 16 bits in the immediate operand.

In the MIPS, there are similar compare instructions but they are termed *Set On Condition* operations. Their function is that a register is set to true or false depending on whether a condition (condition) is met when comparing two values.

7. **Unconditional jumps in Nios II.** The instructions `br` or `jmp` calculate an address which is then written to the program counter. This means that the next instruction fetched is from the location in memory indicated by the calculated address.

   The instruction `jmp` has a constant address operand that should be in a register. The contents of the source register is copied unchanged to the program counter.

   The instruction `br` adds a constant offset to the current value in the program counter. The constant number is indicated in the `br` instruction and can be both positive and negative. Positive offsets give jump ahead, and negative offsets give jump back in the program. The offset is indicated by 16 bits.

8. **Conditional jumps in Nios II.** The following conditional jump instructions are available: `beq, bne, bge, bgeu, bgt, bgtu, ble, bleu, blt, bltu`.

   Conditional jump instructions evaluate the same kind of conditions as `cmp` instructions. If the condition is fulfilled, then the jump is taken (the program counter is changed), otherwise the program continues execution with the instruction located immediately after the conditional branch instruction in memory.

   There is only one addressing mode: both of the operands being compared must be in registers. It is not possible to use a conditional branch instruction for comparing a register content with a constant value, as the instruction `cmpeqi` does. Such comparisons requires a `cmp` instruction followed by a conditional branch instruction.

Exception: Comparing with register r0 can be seen as comparing with value 0.

Example of a conditional branch instruction:

```
1  bgt r8, r9, LABEL # jump to the LABEL on r8 > r9
```

The marker LABEL is written in the program as `LABEL:` at the beginning of a line. The translation program (assembler) that translates assembly into binary machine code calculates the distance from the address immediately following the conditional branch instruction to the marker LABEL, and provides it as the constant offset in the branch instruction. Remember that the distance is counted as negative if the LABEL is before `bgt r8, r9, LABEL` in the program (and as positive if the LABEL available for `bgt r8, r9, LABEL`).

9. **A program with a conditional branch.**

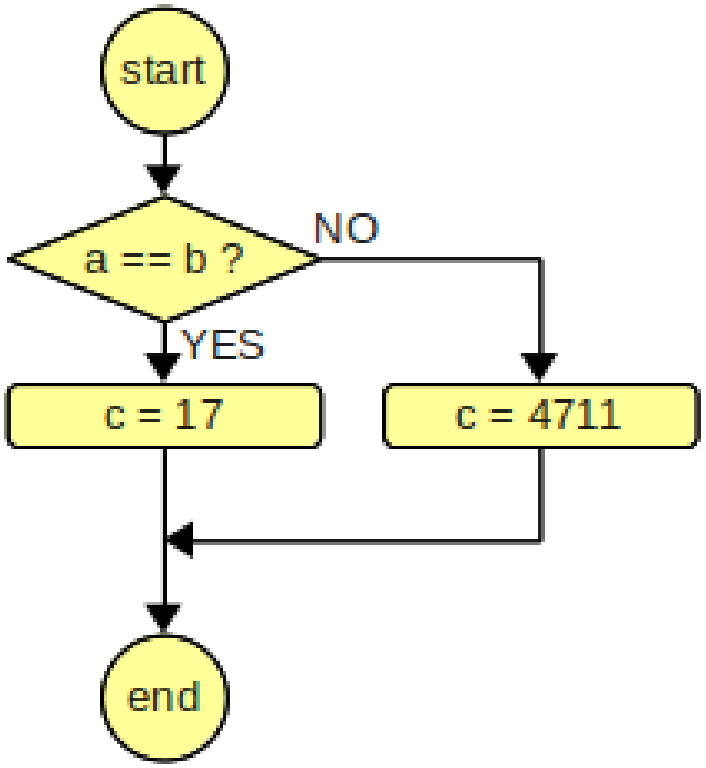

Figure 5: Flowchart for the conditional branch program.

(a) Flowchart: See Figure 5.

(b) Declarations are translated the same way as before to

```
.data       # indicates that variable declarations come after .data
.align 2    # needed for a word remains on a valid address
a: .word 0  # reserves space for a word (4 bytes) with content 0
b: .word 0
c: .word 0
```

Program line

```
if (a == b) / * then * / c = 17; else c = 4711;
```

is then translated differently according to sub task requirements.
To translate the program using only conditional branch instructions.

```
.text            # indicates that code comes after the .text
.align 2         # needed for a word remains on a valid address
movia r8, a      # move the address of the variable a in r8
ldw r9,0 (r8)    # read the contents of the variable a from memory to r9
movia r8, b      # move the address of the variable b to r8
ldw r10,0 (r8)   # read the contents of the variable b from memory to r10
bne r10, r9, L1  # if the contents are different, skip to L1
movi r12,17      # then-part comes here, prepare to add 17 to c
goto L2          # skip past the else part
L1:              # the else part comes here
movi r12,4711    # prepare to add 4711 to c
L2:              # following program code is always executed
movia r8, c      # move the address of the variable c in r8
stw r12,0 (r8)   # write the final result to variable c in memory
```

Reminder: In the Nios II assembly language, the # character implies that the rest of the line is a comment.

(c) To translate the program using `cmp` instruction.

```
.text              # indicates that code comes after the .text
.align 2           # needed for a word remains on a valid address
movia r8, a        # move the address of variable a in r8
ldw r9,0 (r8)      # read content of variable a from memory to r9
movia r8, b        # move the address of variable b in r8
ldw r10,0 (r8)     # read contents of variable b from memory to r10
cmpeq r11, r10, r9 # compare if contents of variables a and b are equal
beq r11, r0, L1    # if (false) contents were different, skip to L1
movi r12,17        # then-part comes here, prepare to add 17 to c
br L2              # skip past the else part
L1:                # the else part comes here
movi r12,4711      # prepare to add 4711 to c
L2:                # following part of the program code is always executed
movia r8, c        # move the address of variable c to r8
stw r12,0 (r8)     # write the final result to variable c in memory
```

Reminder: In the Nios II assembly language, the # character implies that the rest of the line is a comment.

10. **A program with a loop.**

The original program was:

```c
int n;
int sum;
int i;

sum = 0;
for (i = 1; i <= n; i = i + 1)
{
sum = sum + i;
}
```

See the flow chart in Figure 6.

(a) For the while loop version, declarations at the beginning are retained, the rest is replaced with the following.

```c
sum = 0;
i = 1;
while (i <= n)
{
    sum = sum + i;
    i = i + 1;
}
```

(b) For the if and goto version, declarations at the beginning are retained, the rest is replaced with the following.

```c
sum = 0;
i = 1;
loop: if (i > n) goto endofloop;
/ * note: condition has been inverted. * /
sum = sum + i;
i = i + 1;
goto loop;
endofloop: / * program continues. * /
```

(c) The corresponding program in Nios II assembly.

```
# This is a directive to the assembler.
# Everything that follows the .data statement is
# placed in a section of memory that is reserved for data.
# Data must be in read and write memory (RAM).
.data
sum: .word 0      # Reserving space for variable sum.
in: .word 0       # Reserving space for variable i.
n: .word 17       # Reserving space for variable n.

# This is also an assembler directive.
# Everything that follows the .text statement is
# placed in a section of memory that is reserved for program code.
# Program code can be in read-only memory.
```

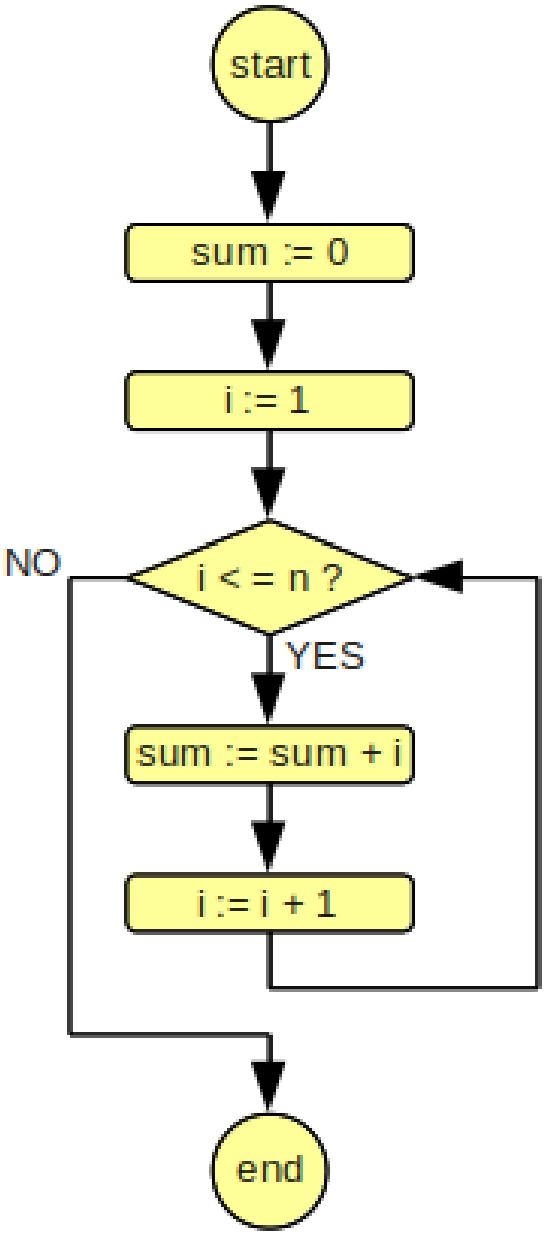

Figure 6: Flowchart for the loop program

```
14  .text
15  movia r8, sum      # point out the first variable.
16  stw r0,0 (r8)      # zero sum, r0 contains zero
17  movi r9,1          # prepare to put in the first index
18  stw r9,4 (r8)      # i is set to the first index
19  L1:
```

```
20  ldw r10,4 (r8)     # read in
21  ldw r11,8 (r8)     # read n
22  bgt r10, r11, L2   # if i> n, jump to L2
23
24  ldw r11,0 (r8)     # read sum
25  add r11, r11, r10  # add sum
26  stw r11,0 (r8)     # save the new value of sum
27
28  ldw r10,4 (r8)     # read in
29  addi r10, r10,1    # increase in by 1
30  stw r10,4 (r8)     # save the new value of in
31
32  br L1              # go to the beginning of the loop.
33  L2:
```

Assembly program is written in a particular style. In this programming style, each C statement is translated separately, to one or more assembler instructions. The variables are stored in memory. Each variable is read from the memory before each update, and the new value is written back to memory immediately after each update. An advantage of this programming style is that the variables are declared in sequence one after the other. The programmer has taken advantage of this by placing the address of the first variable in a register r8, which then does not change. Then each variable is read and written with load and store instructions that use register r8 and an offset - 0 (r8), 4 (r8) or 8 (r8) - depending on the variable being referenced. The programming style is meant to resemble the assembly code as a C compiler produces without any optimization. The style has advantages especially when troubleshooting. Since every C program line is translated separately, it is easy to connect a block of assembly code to the corresponding C program line. It is of course possible to shorten this assembly program, that is, optimize it. One possibility, which may seem obvious, is to add the variables i and sum in separate registers and to not write their values to memory until after the entire loop is completed, after the state L2. The C compiler can do this, but only if the user activates optimizations.