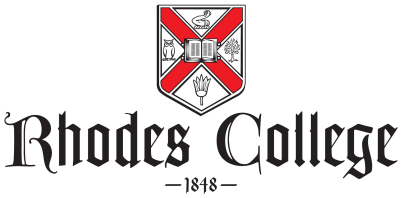


Introduction to NIOS Assembly Language

Contents

1 Hardware Environment

We will be using a CPU implemented on the FPGA devices for the remaining assignments in the course. It is called the NIOS CPU and is implemented by Altera/Intel. Similar to our earlier labs, the CPU design is specified within an Altera Quartus project. To use the CPU, we use another application, the *Altera Monitor Program* to interface with the CPU hardware and to download our programs onto it. We will not be using the book's CPU (MARIE), as it is too simple and doesn't represent a real-world environment. This document will fill-in for the NIOS-specific components of the textbook.

The NIOS CPU is derived from the well-known MIPS architectures. It is a reduced-instruction set architecture (RISC). This means that rather than supporting a wide-range of addressing modes (see below), NIOS provides only a few that makes the hardware implementation simpler, and therefore faster. The primary impact on us is that the NIOS is a *load/store* architecture.

Data on the NIOS system can reside either in main memory or within registers directly on the CPU. Most assembly language instructions work with register data. For example, an addition operation takes two values located in registers and places the result into another register. Any data that is located in main memory must first be brought into a register with a *load* operation and the result may be written back to main memory with a *store* operation.

Here are some basic characteristics of the NIOS CPU system:

- RISC - load/store based CPU
- System clock runs at 50MHz (50,000,000 cyc/sec)
- On-chip memory is 4KB (can be extended to 128MB)
- 32-bit word length (primary data size is 4 bytes)
- Memory addresses are 32-bits
- Special *Program Counter* (**PC**) register contains address of next instruction to execute
- Signed integers use 2's complement representation
- By default, multiplication and division hardware is omitted
- Floating-point values are not supported
- Devices (switches, LEDs, SSDs, etc.) accessed through *Parallel I/O Ports* (PIO)

Before we get into the specifics of the NIOS assembly language, we need to define some properties of the hardware as well as a means for abstractly describing the behavior of each assembly instruction.

1.1 Registers

The NIOS CPU has 32 registers that are used for short-term memory directly on the CPU itself. These registers are implemented as a set of D flip-flops, each able to store a 32-bit value. Registers are very fast and can be accessed without delay by the CPU. In contrast, accessing main memory may take 70-100 cycles before the data can be used.

The registers on the NIOS are identified by their names `r0..r31`. Some registers have special purposes and are reserved by convention for specialized purposes. These registers may be referenced by special names to make their usage clear (such as `sp` for the stack pointer, but they are just aliases for the actual register (`r31` in the case of the stack pointer).

For the moment, we won't focus on the detailed conventions for the registers, but we will observe some broad rules on register usage. The compiler for a high-level language produces assembly language as its output. It is important for all compilers to follow some basic rules about register usage, so that the code produced by different compilers can function together. (i.e. we expect that `cout` behaves correctly with our program, regardless of the compiler used to build the system libraries). We will deal with this more in-depth later, but for now let's work within the following boundaries:

- `r0` – `zero` - register zero is a special register. It cannot be written to and always returns a zero when read. This will be useful later.

- `r1..r23` – these registers are fair game and can be used to hold any data that is needed to complete a computation.
- `r24..r31` – these registers are reserved by convention. They are used to support functions in high-level languages, interrupts, and debugging, etc. They are not to be used at this point.

There are also 6 specific control registers that are used to support high-level operations that are usually performed by an operating system (for paging, interrupts, status, etc.) We are ignoring them.

1.2 Addressing Modes

Memory is accessed using load and store instructions. No other assembly instructions may reference data stored in memory. The load and store instructions take a register as one operand and a memory location as another. A load operation will read the data from the address in memory and load it into the register. A store operation goes in the opposite direction – the value in the register is written to the location in memory.

There are multiple ways of specifying the location of data within the system. Each mechanism is called an *addressing mode*. The NIOS architecture supports five distinct addressing modes:

- **Immediate** – the instruction contains the value to be used as an operand. This is usually a 16-bit value. In an expression like: `x = y + 6`, the value 6 would be an immediate operand.
- **Register Direct** – the operand is stored in a register. In the expression `x = y + 6`, if `y` were stored in register `r4`, then `y` would be accessed using the register direct addressing mode.
- **Register Indirect** – the operand's *address* is stored in a register. In the expression `x = *y + 6`, `y` is a pointer value. If the pointer value (i.e. the memory address) were stored in register `r4` then `*y` would be accessed using the register direct addressing mode.
- **Absolute** – the operand's *address* is specified as an absolute, immediate value. This is used if you would like to specify the exact memory address to reference (i.e. memory address `0x1000`).
- **Displacement** – the operand's *address* is computed as the sum of a value in a register and a 16-bit immediate displacement value. This is typically used with a data structure. Imagine we have a 3-D point with x, y, z values located in memory at address `0x1000`. If each value is an integer, then x would be at `0x1000`, y at `0x1004`, and z at `0x1008`.

  If the address `0x1000` were located in register `r4`, then we could the value of y would be located at `4(r4)` and `8(r4)`. I.e. we add the immediate value (4 or 8) with the register that points to the beginning of the data structure.

It is important to understand the displacement addressing mode because it is the form for all of the register indirect, absolute, and displacement addressing modes. Imagine that we wished to access the memory location `0x1004` in each of the register indirect, absolute, and displacement modes. We can express each of these modes using the displacement syntax:

```
disp(rX)
```

The *disp* is an 16-bit displacement that is added to register *rX* to compute the memory address. The displacement is always first and outside of the parens, the register is always second and placed within the parens.

Table 8-1: I-Type Instruction Format

Bit Fields														
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										OP					

Table 8-2: R-Type Instruction Format

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					OPX
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPX										OP					

Table 8-3: J-Type Instruction Format

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IMM26															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM26										OP					

Figure 1: Instruction formats for NIOS assembly instructions

Absolute: To produce an absolute address, we need to find a register that always produces zero. Luckily, we have `r0`. This is the displacement mode syntax for an absolute address:

```
0x1004(r0)
```

Displacement: Place the value `0x1000` in register `r4`, and use 4 as the displacement value. This is the displacement mode syntax:

```
4(r4)
```

1.3 Instruction Formats

All assembly-language instructions are encoded as 32-bit strings. The specific encoding depends on the instruction and its operands. We'll look at some examples later, but for now we just need to know that NIOS has three basic templates for instructions. All assembly language instructions must fit into one of these three templates shown in Figure 1.

The main point to observe these in these instructions is that the format must contain all of the information that is needed to perform the operation. For example, an add operation, such as: `z = x + y` needs to encapsulate that a) we want to add two things (the operation) b) the things we want to operate on are `x` and `y` (the operands) and lastly c) the result should be stored into `z`.

1.4 Register Transfer Notation

Now we know about the basic structure of the NIOS CPU, the names of registers, the different ways to refer to data (addressing modes), and binary encoding format for assembly instructions. When we look at the assembly instructions, we need to have a precise way for characterizing the behavior of each one. A single assembly instruction may do several things during execution and the way we express these is by using a *register transfer language* (RTL). This is often also referred to as an RTN, or register transfer notation. Either RTL or RTN can be used interchangeably. A list of valid RTN expressions can be seen in Figure 2.

Table 8-8: Notation Conventions

Notation	Meaning
X ← Y	X is written with Y
PC ← X	The program counter (PC) is written with address X; the instruction at X is the next instruction to execute
PC	The address of the assembly instruction in question
rA, rB, rC	One of the 32-bit general-purpose registers
prs.rA	General-purpose register rA in the previous register set
IMMn	An n-bit immediate value, embedded in the instruction word
IMMED	An immediate value
X_n	The nth bit of X, where n = 0 is the LSB
$X_{n..m}$	Consecutive bits n through m of X

Notation	Meaning
0xNNMM	Hexadecimal notation
X : Y	Bitwise concatenation For example, (0x12 : 0x34) = 0x1234
σ(X)	The value of X after being sign-extended to a full register-sized signed integer
X >> n	The value X after being right-shifted n bit positions
X << n	The value X after being left-shifted n bit positions
X & Y	Bitwise logical AND
X | Y	Bitwise logical OR
X ^ Y	Bitwise logical XOR
~X	Bitwise logical NOT (one's complement)
Mem8[X]	The byte located in data memory at byte address X
Mem16[X]	The halfword located in data memory at byte address X
Mem32[X]	The word located in data memory at byte address X
label	An address label specified in the assembly file
(signed) rX	The value of rX treated as a signed number
(unsigned) rX	The value of rX treated as an unsigned number

Figure 2: Register Transfer Notation for NIOS Instructions

2 Assembly Language

Next we are going to look at specific assembly language instructions. A *program* written in NIOS assembly language consists of a sequence of instructions. In contrast to higher-level programming languages, there are no built-in mechanisms for even rudimentary programming concepts such as conditionals, loops, or functions. All of these constructs must be built using the instructions provided by the *instruction set architecture* or ISA.

We will look at how to map simple program excerpts from a higher-level language like C/C++ to NIOS assembly in the next section. Before we do, we need to understand the different types of instructions, how we write them, and what they do. NIOS assembly instructions can be roughly grouped into the following categories:

- **Load/Store** – fetch data from main memory into a register (load) or write data from a register into main memory (store)
- **Arithmetic** – perform math and logic operations, such as addition, subtraction, AND-ing, OR-ing, shifting, etc.
- **Move** – copy a word from one register to another, or immediate (constant) data into a register. Move operations never work on locations main memory.
- **Comparisons** – compare the contents of a register and either another register or an immediate (constant) value.
- **Branch/Jump** – alters the program flow either based on a condition (branch) or unconditionally (jump)
- **Functions/Subroutines** – the `call` and `ret` instructions are specialized jump instructions that support function calls.

The full list of instructions supported by the NIOS CPU are listed in the `NIOS II Classic Processor Reference Guide` published by Altera and posted on Moodle. You should be familiar with the reference manual and able to use it to answer questions about instructions that you may need to complete your assignments. Note that the # symbol is used for comments in NIOS assembly language.

2.1 Arithmetic/Logic Instructions

We'll start by looking at a few basic math instructions to get a sense for how they work. First let's see how to add two numbers using assembly language.

2.1.1 add

The reference sheet for the `add` instruction is listed below. As can be seen from the sheet, `add` takes three registers as parameters in the following format: `add rC, rA, rB`. The reference sheet provides a description of the instruction using RTN. The RTN for `add` is given as: $rC \leftarrow rA + rB$. `rA, rB,` and `rC` denote any of the general purpose registers on the NIOS, but we're limiting ourselves to `r1..r23`. The RTN says that registers `rA` and `rB` are added together and the result is stored in `rC`. To perform an addition using the `add` instruction, our operands must all be in registers and the output is stored in another register.

2.1.2 addi

Other information on the reference sheet gives us a usage example, shows us the instruction format type, and in this case describes how to check for carry and overflow conditions. The NIOS processor does not keep a `C` or `V` register for these conditions, extra processing must be done to check the validity of the addition. Our assembly programs do *not* need to check explicitly for carry or overflow.

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 rC, rA, rB              # The original add operation
cmpltu rD, rC, rA           # rD is written with the carry bit

add rC, rA, rB              # The original add operation
bltu rC, rA, label          # Branch if carry generated
```

Overflow Detection (signed operands):

An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown in the following code:

```
add rC, rA, rB              # The original 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              # Combine comparisons
blt rD, r0,label            # Branch if overflow occurred
```

Exceptions: None

Instruction Type: R

Instruction Fields:

`A` = Register index of operand rA

`B` = Register index of operand rB

`C` = Register index of operand rC

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					0x31						0					0x3a						

Sometimes we would like to perform an addition with a value that is known precisely at compile time. In a loop, for example, it is common to execute a statement like this: `i = i + 1;` . To do this, we use the `addi` or *add immediate* instruction. As can be seen from the reference sheet, we specify an immediate value as the last operand.

It's important to understand the meaning of the RTN for this instruction. The RTN tells us that the `addi` performs the following action: $rB \leftarrow rA + \sigma(IMM16)$. There are two things that are important here. The first is that the instruction permits an `IMM16`. That is, the immediate value is treated as a 16-bit value. Why? To understand this, we need to look at the instruction format for `addi`. It's listed as an I-type instruction on the sheet. Looking back at Figure 1, we can see that 16 bits of the instruction are reserved for an immediate value.

addi **add immediate**

Operation: rB ← rA + σ (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:

```
addi rB, rA, IMM16          # The original add operation
cmpltu rD, rB, rA           # rD is written with the carry bit

addi rB, rA, IMM16          # The original add operation
bltu rB, rA, label          # Branch if carry generated
```

Overflow Detection (signed operands):

An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown in the following code:

```
addi rB, rA, IMM16          # The original add operation
xor rC, rB, rA              # Compare signs of sum and rA
xorhi rD, rB, IMM16         # Compare signs of sum and IMM16
and rC, rC, rD              # Combine comparisons
blt rC, r0,label            # Branch if overflow occurred
```

Exceptions: None

Instruction Type: I

Instruction Fields: `A` = Register index of operand rA

`B` = Register index of operand rB

`IMM16` = 16-bit signed immediate value

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																0x04					

So, the immediate value used for the addition must be stored *within the instruction itself*. Since all instructions are 32-bits, any immediate data that is encoded into the instruction must be smaller than 32-bits. What happens if we try to do this?

```
addi   r4, r5, 0x10000
```

The value `0x10000` can only be represented in a number that is 17 or more bits! This number cannot be used in an immediate instruction. To perform additions with larger values, they would need to be put into a register first, then added with the traditional (non-immediate) `add` instruction.

Next, let's look at the σ operator in the `addi` RTN. This specifies that our immediate value will be *sign-extended*. We know that the immediate is represented as a 16-bit value, but that the value in the register is a 32-bit value. What happens the high-order bits from our immediate parameter? Let's imagine that we our 32-bit register (say, `r5`) contains the number 2 using signed 2's complement representation and we want to add 1 to it:

```
addi   r5, r5, 1
```

This performs the binary addition like this:

```
  0000 0000 0000 0000 0000 0000 0000 0010    (2)
+                     0000 0000 0000 0001    (1)
-------------------------------------------------
```

If we assume that all of the high-order bits of the immediate value are zeros, we do okay and it looks like this:

```
  0000 0000 0000 0000 0000 0000 0000 0010    (2)
+ 0000 0000 0000 0000 0000 0000 0000 0001    (1)
-------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0011    (3)
```

What happens if we have a negative number, like -1? Recall that the two's complement representation for -1 is all ones. Let's try again but this time with a -1:

```
addi   r5, r5, -1
```

This gives us the following addition in binary:

```
  0000 0000 0000 0000 0000 0000 0000 0010    ( 2)
+                     1111 1111 1111 1111    (-1)
-------------------------------------------------
```

If we follow the same rule above, and just replace the high-order bits with zeros, we get the following:

```
  0000 0000 0000 0000 0000 0000 0000 0010    (    2)
+ 0000 0000 0000 0000 1111 1111 1111 1111    (65535)
-----------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0011    (65537)
```

This isn't good. Our negative 16-bit value turns into a positive 32-bit value if we replace the high-order bits with zero. To solve this, we perform a *sign-extension* instead of just using zeros. To sign extend a shorter number into a longer one, we only need to look at the highest-order bit of the shorter number. In the case of 1, this is a zero and in the case of -1, this is a 1. I.e. if the number is positive, we fill with zeros and if negative, fill with ones. If we follow this rule, then the 32-bit addition looks like this:

```
  0000 0000 0000 0000 0000 0000 0000 0010    (    2)
+ 1111 1111 1111 1111 1111 1111 1111 1111    (   -1)
-----------------------------------------------------
1 0000 0000 0000 0000 0000 0000 0000 0001    (    1)
```

Here we see that the addition results in a carry-out, but the 32-bit result is correct.

2.2 Load/Store Instructions

All memory accesses using assembly language must either use a load instruction (for reads) or a store instruction (for writes). Both instructions share syntax, in that the first operand is a register and the second is a memory location specified using the displacement addressing mode.

2.2.1 ldw

Looking at the reference sheet for the `ldw` instruction, we can see that there are two variants: `ldw` and `ldwio`. On our FPGA, some of the devices are *memory-mapped*, which means that access to the device is done by reading or writing to a special address in memory. Reading or writing to a device memory address **must be performed using the** `io` **variant**. Most normal memory accesses will use the non-io version.

Again, to best understand the operation of the load instruction, we need to consider the RTN:
$rB \leftarrow Mem32[rA + \sigma(IMM14)]$. In English, this reads that "the value in the `rA` register is added to the sign-extended 14-bit displacement to form a memory address. A 32-bit value is read from memory, starting at this address and the value is stored into `rB`."

Our displacement values are limited to 14-bit values and they may be signed (i.e. negative displacements are okay). When we put all of this together, we can see that a displacement may be in the range $-2^{13} \ldots 2^{13} - 1$ or `-8192..8191`. If we need to use a larger displacement, we would just perform an addition or subtraction on the `rA` value directly.

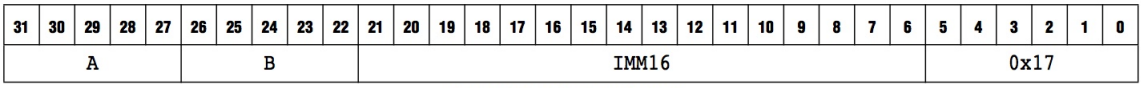
The reference sheet also says that "*the effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.*" You should read "is undefined" as "it won't work". This is a common behavior to many CPU architectures. It is more efficient to only support reading multi-byte values from memory on word boundaries.

Information in memory on the NIOS system is stored in 32-bit chunks (words). We can tell if an address is word-aligned if it's evenly divisible by 4. So, address 1000 is word-aligned, but 1001 is not. The issue is that memory is organized into words and that requesting a 32-bit value starting at 1001 is troublesome. This would require fetching 3 bytes from the word at address 1000 (bytes 1001, 1002, and 1003) and 1 byte from the next word at address 1004. Since memory is stored and accessed in word chunks, this read would require the fetch of two words and then the data would all have to be stitched together and put into a register. This is a lot of hassle, so most CPUs just say "you have to access memory on word boundaries".

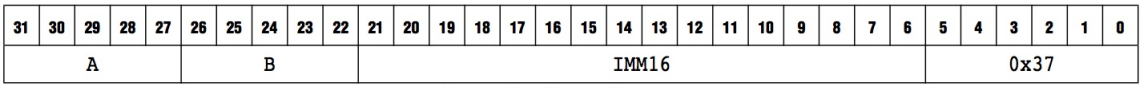
There are several variants of the load instruction for accessing data of differing sizes. For example, `ldb` would load a single byte, `ldbio` reads a byte from a device address, `ldh` would load a half-word (16-bits), etc.

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*.
Exceptions:	Supervisor-only data address Misaligned data address TLB permission violation (read) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type:	I
Instruction Fields:	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value



Instruction format for `ldw`



Instruction format for `ldwio`

2.2.2 stw

The `stw` instruction just performs the opposite of the `ldw` instruction. The instruction format is just the same, but the data in the register is written out to the address specified using the displacement addressing mode. The same caveats and behaviors described above for the load operation apply for stores as well.

There are several variants of the store instruction for accessing data of differing sizes. For example, `stb` would store a single byte, `stbio` writes a byte to a device address, `sth` would store a half-word (16-bits), etc.

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.

2.3 Logical Operation Instructions

We're going to take a look at a few logical operation instructions because they are representative of a larger class of useful instructions. We'll focus on a bitwise-logical AND operation and it's variants. Other logical operations exist (OR, NOT) and are described in the reference manual.

2.3.1 and

The `and` instruction takes two 32-bit values from registers and performs an AND operation between each corresponding bit, then stores the result into the destination register. For example, consider the statement:

```
and r5, r3, r4
```

If `r3` contains decimal 3 (`00..11`) and `r4` contained decimal 2 (`00..10`), then the bitwise AND would be decimal 2 `00..10`. The lowest bits are ANDED together (1 and 0), the next lowest (1 and 1), and so on. The result is stored into `r5`.

and — **bitwise logical and**

Operation:	rC ← rA & rB
Assembler Syntax:	`and rC, rA, rB`
Example:	`and r6, r7, r8`
Description:	Calculates the bitwise logical AND of rA and rB and stores the result in rC.
Exceptions:	None
Instruction Type:	R
Instruction Fields:	`A` = Register index of operand rA
	`B` = Register index of operand rB
	`C` = Register index of operand rC

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					0x0e						0					0x3a						

2.3.2 andi

Unsurprisingly, there is an immediate version as well, `andi`. This takes an immediate 16-bit value. Since AND is a different operation than addition, we should see how we deal with the interaction of a 16-bit immediate and a 32-bit

register value. Let's assume we have an instruction like this: `andi r4, r5, 0xFF`. How will we find out what `andi` does? Let's look at the RTN:

$$\texttt{rB} \leftarrow \texttt{rA}\&(\texttt{0x0000} : \texttt{IMM16})$$

Now we can see that the 32-bit value to AND with the register is constructed by concatenating 16-bits of zeros to the 16-bit immediate value. Our example actually computes the bit-wise AND of the contents of `r5` and `0x0000FFFF` and stores the result in `r4`. This permits us to AND immediate values with the low-order 16-bits in the register, but what about the high-order 16-bits?

andi **bitwise logical and immediate**

Operation:	rB ← rA & (0x0000 : IMM16)
Assembler Syntax:	`andi rB, rA, IMM16`
Example:	`andi r6, r7, 100`
Description:	Calculates the bitwise logical AND of rA and (0x0000 : IMM16) and stores the result in rB.
Exceptions:	None
Instruction Type:	I
Instruction Fields:	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit unsigned immediate value

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																0x0c					

2.3.3 andhi

By looking at the RTN for the `andhi` instruction we see the following:

$$\texttt{rB} \leftarrow \texttt{rA}\&(\texttt{0x0000} : \texttt{IMM16})$$

Here we can see that `andhi` operates much like `andi`, but it works with the highest 16-bits in the word, rather than the lower 16-bits. By using both `andi` and `andhi` together, we can use immediate values to work on an entire 32-bit value.

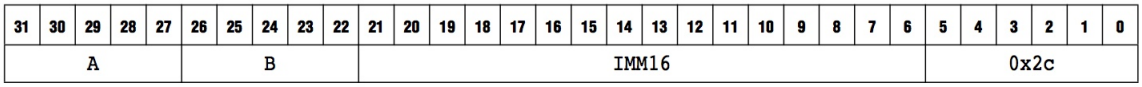
This occurs frequently when we are using a 32-bit value to represent several smaller values. For example, if we were to store four 8-bit values in a single 32-bit register we can isolate the values by using a technique called *masking*. If we had four 8-bit values, `0x01, 0x02, 0x03, 0x04`, stored in register `r4` as a 32-bit value, `0x01020304`. We could isolate each value using the following sequence of instructions:

```
andi  r5, r4, 0x00FF  # isolates 0x04
andi  r6, r4, 0xFF00  # isolates 0x03
andhi r7, r4, 0x00FF  # isolates 0x02
andhi r8, r4, 0xFF00  # isolates 0x01
```

At the end of this sequence, each 8-bit value would be extracted to registers `r5, r6, r7`, and `r8`. Each value hasn't been moved within the word, just isolated. That is to say, that `r8` contains `0x01000000` and not `0x000000001`. If we would like `r8` to represent the value `0x01`, then we need some way to "move over" the bits into the right place.

andhi **bitwise logical and immediate into high halfword**

Operation:	rB ← rA & (IMM16 : 0x0000)
Assembler Syntax:	`andhi rB, rA, IMM16`
Example:	`andhi r6, r7, 100`
Description:	Calculates the bitwise logical AND of rA and (IMM16 : 0x0000) and stores the result in rB.
Exceptions:	None
Instruction Type:	I
Instruction Fields:	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit unsigned immediate value



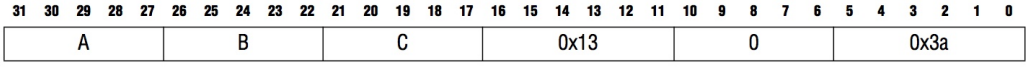
2.3.4 sll

To move bits around within a register, we can use a *shift* instruction. There are instructions to perform both left and right shifts, along with register and immediate variants. Recall that if we shift a binary number left by k bits, it has the same effect as multiplying the original value by 2^k. Similarly, shifting a binary number right by k bits is akin to dividing by 2^k, discarding the remainder.

The `sll` instruction performs a *shift-logical left*. The reference sheet tells us that this instruction shifts the value in the `rA` register by the number of bits stored in the `rB` register and stores the result in `rC`. Since we are shifting to the left, we will be discarding values in the higher-order bits and must replace them with new values inserted at the right (low-order) end. The `sll` instruction states that the new values inserted are always zero. This makes sense since 0001 shifted left by 4 bits is 1000, which is equivalent to $1 \times 2^4 = 16 = 1000$.

sll **shift left logical**

Operation:	rC ← rA << ($rB_{4..0}$)
Assembler Syntax:	`sll rC, rA, rB`
Example:	`sll r6, r7, r8`
Description:	Shifts rA left by the number of bits specified in $rB_{4..0}$ (inserting zeroes), and then stores the result in rC. `sll` performs the << operation of the C programming language.
Exceptions:	None
Instruction Type:	R
Instruction Fields:	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC



2.3.5 srli

When shifting to the right, there are some additional subtleties. We'll look at two different mechanisms for performing right shifts using immediate values.

First up is `srli`. A quick look at the reference sheet shows that this instruction works mostly the same as `sll`, but uses an immediate value and shifts to the right instead of the left. The values inserted from the left side are zeros. In some cases, this works fine. If the source register contains the 32-bit value `0x00000010` (16_{10}) and we shift right by 4, then we get `0x00000001` (1_{10}). This works well with our notion that shifting right is equivalent to division (4-bit shift is equal to dividing by $2^4 = 16$).

However, we run into trouble when shifting right when we have a negative value. Imagine we have a signed 8-bit value `1111 1110` (-2_{10}). What happens when we shift 1-bit to the right (i.e. dividing by 2)? If we use a logical shift, we would shift in zeros from the left, giving us: `0111 1111` (127_{10}). Luckily there is a fix.

srli **shift right logical immediate**

Operation:	rC ← (unsigned) rA >> ((unsigned) IMM5)
Assembler Syntax:	`srli rC, rA, IMM5`
Example:	`srli r6, r7, 3`
Description:	Shifts rA right by the number of bits specified in IMM5 (inserting zeroes), and then stores the result in rC.
Usage:	`srli` performs the unsigned >> operation of the C programming language.
Exceptions:	None
Instruction Type:	R
Instruction Fields:	`A` = Register index of operand rA `C` = Register index of operand rC `IMM5` = 5-bit unsigned immediate value

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					0					C					0x1a						IMM5					0x3a						

2.3.6 srai

Instead of using the `srli` instruction for division by powers of two, we can use the `srai`, or *shift right arithmetic immediate*, instead. To see how this fixes the problem we can look at the `srai` reference sheet. In the description, it tells us that the instruction performs the shift, but rather than shifting in zeros from the left, it duplicates the sign bit. Let's see if this fixes our problem.

If we go back to our example above, we want to shift `1111 1110` (-2_{10}) right by one bit. If we duplicate the sign-bit (a 1 because it's negative), we get this: `1111 1111` (-1_{10}), which gives us the answer we want.

Lastly, we need to pay special attention to the immediate shift instructions. They are all categorized as R-type instructions. Rather than 16-bit immediate values used by the `andi/andhi` instructions above, shift instructions are only permitted a 5-bit immediate operand. As it turns out, this is not a big problem since $2^5 = 32$ and we can only shift left or right a maximum of 32 bits.

srai **shift right arithmetic immediate**

Operation:	rC ← (signed) rA >> ((unsigned) IMM5)
Assembler Syntax:	`srai rC, rA, IMM5`
Example:	`srai r6, r7, 3`
Description:	Shifts rA right by the number of bits specified in IMM5 (duplicating the sign bit), and then stores the result in rC.
Usage:	`srai` performs the signed >> operation of the C programming language.
Exceptions:	None
Instruction Type:	R
Instruction Fields:	`A` = Register index of operand rA `C` = Register index of operand rC `IMM5` = 5-bit unsigned immediate value

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					0					C					0x3a						IMM5					0x3a						

2.4 Pseudo Instructions

Sometimes the objective of the CPU design and the needs of low-level programmers diverge. In hardware design, the goal is to provide the maximum functionality with the minimum complexity. Whereas, as software developers (and *especially* with assembly language), we care about clarity and readability of our code as well.

The hardware manufacturer will often provide a set of *pseudo instructions* that make it easier to write an assembly program, but is more efficiently performed using existing instructions. As a programmer, you are free to write your program with these pseudo instructions and the assembler will replace them with their equivalent operations when generating the executable program.

2.4.1 mov

Let's look at `mov`, first. This instruction just copies the contents of one register to another. Rather than having to implement a new instruction that performs this task, we can be creative about how we use the `add` instruction. If we want to move the contents of `r8` to `r9`, we can just add and use the zero register: `add r9, r8, r0`. This accomplishes the same task as the `mov` instruction, using the existing add instruction. When you run your programs in the monitor program, you would see that the `mov` instruction was replaced with an `add` in this form.

mov **move register to register**

Operation:	rC ← rA
Assembler Syntax:	`mov rC, rA`
Example:	`mov r6, r7`
Description:	Moves the contents of rA to rC.
Pseudo-instruction:	`mov` is implemented as `add rC, rA, r0`.

2.4.2 movi

When moving an immediate value into a register, we can use a similar trick. The assembler replaces any instruction like: `movi r8, 128` with an `addi` instruction that does the same thing: `addi r8, r0, 128`

The immediate value used with the `movi` has the same limitation as the `addi` immediate, i.e. it can be at most a 16-bit value.

movi — **move signed immediate into word**

Operation:	rB ← σ (IMMED)
Assembler Syntax:	`movi rB, IMMED`
Example:	`movi r6, -30`
Description:	Sign-extends the immediate value IMMED to 32 bits and writes it to rB.
Usage:	The maximum allowed value of IMMED is 32767. The minimum allowed value is –32768. To load a 32-bit constant into a register, refer to the `movhi` instruction.
Pseudo-instruction:	`movi` is implemented as `addi rB, r0, IMMED`.

2.4.3 movhi

The `movhi` instruction allows to explicitly set the 16 high-order bits with an immediate value. The instruction `movhi r8, 0xFFFF` will set the highest 16 bits in `r8` to all ones. We can't rely on the `add/addi` tricks from before. To do this, the assembler uses the `orhi` instruction. This works the same as the `andhi` instruction, but uses the bitwise OR instead of AND. The pseudo instruction is rewritten as: `orhi r8, r0, 0xFFFF` This performs a bitwise-OR with the zero register (all-zeros) and stores the result in the high-order bits of `r8`.

If we recall the boolean algebra identity: $x + 0 = x$, we can see that this has the same effect as moving the 16-bit immediate value into the high-order bits of `r8`.

movhi — **move immediate into high halfword**

Operation:	rB ← (IMMED : 0x0000)
Assembler Syntax:	`movhi rB, IMMED`
Example:	`movhi r6, 0x8000`
Description:	Writes the immediate value IMMED into the high halfword of rB, and clears the lower halfword of rB to 0x0000.

Usage:	The maximum allowed value of IMMED is 65535. The minimum allowed value is 0. To load a 32-bit constant into a register, first load the upper 16 bits using a `movhi` pseudo-instruction. The `%hi()` macro can be used to extract the upper 16 bits of a constant or a label. Then, load the lower 16 bits with an `ori` instruction. The `%lo()` macro can be used to extract the lower 16 bits of a constant or label as shown in the following code:
	`movhi rB, %hi(value)`
	`ori rB, rB, %lo(value)`
	An alternative method to load a 32-bit constant into a register uses the %hiadj() macro and the `addi` instruction as shown in the following code:
	`movhi rB, %hiadj(value)`
	`addi rB, rB, %lo(value)`

Pseudo-instruction:	`movhi` is implemented as `orhi rB, r0, IMMED`.

2.4.4 movia

The last thing we need to look at before moving on is at how to move a 32-bit immediate value into a register. There are a few times where this is important, but most often involves a 32-bit memory address that must be placed into memory. In particular, the devices on our DE2 FPGA system will have known, fixed addresses that must be placed into registers before we can perform `ldbio/stbio` operations on them.

To do this, we can use the `movia`, or *move immediate address*, instruction. Once again, we see that `movia` is a pseudo instruction and expands to two instructions, an `orhi` and a `addi`. This is the equivalent of performing both a `movi` for the low 16 bits and a `movhi` for the high 16 bits.

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)`

3 NIOS Expressions

Let's assume that we want to perform a computation that we might find in some high-level programming language like C, C++ or Java:

```
c = a * b + c * d;
```

How do we implement this in assembly language? Well, for starters, there are no variables in assembly language – only locations in memory and registers. The compiler for a high-level language must determine where all of the program variables must exist (not trivial!). For our purposes, we will just make some assumptions about where the data lives and write the instructions based on that. To make things interesting, let's say that all of the variables are located in main memory, starting at memory address 1000, with some initial values as is shown in the following table:

	...	
1012	**5**	d
1008	**4**	c
1004	**3**	b
1000	**2**	a
	...	

Now that we know what computation that we want to perform and how the data is organized in main memory, we can write our assembly code. We are free to use any registers that we wish within the range `r1..r23`.

```
# load addresses of variables into registers
movia r2, 1000   # &a - address of a
movia r3, 1004   # &b - address of b
movia r4, 1008   # &c - address of c
movia r5, 1012   # &d - address of d

# load values
ldw r6, 0(r2)    # a into r6
ldw r7, 0(r3)    # b into r7
ldw r8, 0(r4)    # c into r8
ldw r9, 0(r5)    # d into r9

# perform computation
mul r6, r6, r7   # a * b, store result in r6
mul r8, r8, r9   # c * d, store result in r8
add r6, r6, r8   # (a * b) + (c * d), store result in r6

# save result in &c
stw r6, 0(r4)
```

Here we can see that assembly language is a much simpler and lower-level programming environment. What could be done with a single line of C/C++ requires twelve instructions in NIOS assembly language.

To translate any expression from a high-level language to assembly, we must first understand where things are located (either in memory or register). Any data in memory must be read or written to using is address. Next we break down each step of the expression into the most basic form – one that maps to assembly language instructions. We add load instructions to fetch the necessary data, other instructions to perform the computation, and lastly store out the results if required.

4 Control Flow

At this point, we should have some sense for how we can use assembly instructions to access data in memory and perform some different register-based arithmetic and logical functions. From this, we can take much of a program written in a high-level language and translate it to assembly language as demonstrated in the previous section.

So far, all of the instructions we have seen are meant to be executed in a sequence. If we have an `add` followed by a `stw`, it's not possible for the `add` to "skip over" the `stw`. The reason for this is that the CPU uses a special register, the *program counter*, or PC that contains the address of the next instruction to execute.

Without any intervention, the PC is incremented by 4 during the execution of an instruction (4 = 32-bits, which is the length of a NIOS instruction). So, if our `add` is located at address `1000` and the `stw` is located at `1004`, the store will always directly follow the add. If we go back and check the reference sheets for the store and add, the RTN for neither specifies any modification to the program counter. We now look at some instructions which do modify the program counter and thus alter the flow of execution (the control flow) of our assembly programs.

There are several features of high-level languages that we don't know how to translate to assembly language. We'll start by looking at constructs that affect program flow, beginning with (if/else-if/else) statements.

There are two specific classes of NIOS instructions that can impact the control flow: *unconditional branches* and *conditional branches*. We'll look at unconditional branches first.

4.1 Unconditional Branches

An *unconditional branch* forces the program counter to be set to a new value. These are similar to `goto` statements in a higher-level programming language. The first of these is the `jmp` instruction, which takes a single register as its operand. The instruction `jmp r8` has the effect of setting the PC to the address stored in `r8`. It is important that the address in `r8` refers to an instruction, otherwise the CPU will halt with an error. We can see that the RTN for the `jmp` instruction captures this behavior.

jmp **computed jump**

Operation:	PC ← rA
Assembler Syntax:	`jmp rA`
Example:	`jmp r12`
Description:	Transfers execution to the address contained in register rA.
Usage:	It is illegal to jump to the address contained in register r31. To return from subroutines called by `call` or `callr`, use `ret` instead of `jmp`.
Exceptions:	Misaligned destination address
Instruction Type:	R
Instruction Fields:	`A` = Register index of operand rA

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					0					0					0x0d						0					0x3a						

There is another unconditional branch, `br`, which is similar to the jump, but uses a *label* rather than an address in a register. When writing our assembly programs, it is too cumbersome (even for assembly programmers) to explicitly manage the addresses of every instruction in a program. To simplify things, we can place a name followed by a colon in front of any instruction to note its location in memory. The `br` instruction takes a label as a parameter and jumps to the address of the label. For example:

```
loop:   addi  r1, r1, 1
        br loop
```

This program would execute an infinite loop that continually adds 1 to the value stored in `r1`. A label should start with a letter and may consist of a sequence of other letters or numbers following it.

If the `br` instruction takes a label as a parameter and a label refers to a 32-bit address within a program, how does a 32-bit value get encoded into the 32-bit branch instruction? Good question. The truth is that the label encoded into the branch instruction isn't a full-blown address. In fact, the label is just an offset from the current value of the program counter.

Most branches occur somewhat close to the label that they are jumping to. To handle this, the label is encoded as a 16-bit signed value that represents "how many bytes away" the address to jump to is from the current PC value. Let's take a look at the RTN for `br`:

$$PC \leftarrow PC + 4 + \sigma(IMM16)$$

The way to read this is that the program counter is updated with value PC+4 (i.e. the *next instruction*) added to the sign-extended immediate value encoded in the instruction. If we go back to the add/loop instruction up above, assume that the add instruction is located at memory address 1000 and the branch is at 1004. When we are in the midst of executing the branch, the next instruction is technically at 1008, since the PC should always be referring to the next instruction. The offset for the branch (`loop:`) is -8, since we would need to add -8 back to the PC in order to get to the `addi` instruction at address 1000.

If you find this overwhelmingly confusing, the big picture is that we can't put a 32-bit address into a 32-bit instruction (again). To work around this limitation, we define a branch in terms of the difference between the address of the branch instruction itself (PC+4) and the target label.

br **unconditional branch**

Operation:	PC ← PC + 4 + σ (IMM16)
Assembler Syntax:	`br label`
Example:	`br top_of_loop`
Description:	Transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `br`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions:	Misaligned destination address
Instruction Type:	I
Instruction Fields:	`IMM16` = 16-bit signed immediate value

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
0					0					IMM16																0x06					

4.2 Conditional Branches

In contrast to unconditional branches, a *conditional branch* is a branch that is dependent on some condition. The conditional branches supported by the NIOS CPU operate on two registers and a label. The instruction compares the two register values and branches to the label if the condition is met.

NIOS supports the following conditional branch instructions:

- beq – branches if the two register values are equal

- `bne` – branches if the two register values are not equal
- `blt` – branches if the first register value is less than the second
- `bgt` – branches if the first register value is greater than the second
- `ble` – branches if the first register value is less than or equal to the second
- `bge` – branches if the first register value is greater than or equal to the second

There are other conditional branch instructions that work on unsigned register values, but the basic conditions remain the same.

4.2.1 bne

Let's look at the *branch if not equal* instruction, `bne`. If we have an instruction such as: `bne r8, r9, foo` then the RTN for this instruction tells us that we will branch to the label `foo` if `r8` differs from `r9`, otherwise the program counter is incremented to the next instruction as it normally would.

This is the very essence of a conditional branch. If the condition is *true*, then the program flow is altered, otherwise we continue on as we normally would. This is an important distinction to understand and can often be confusing when constructing conditional logic in assembly language. It bears emphasis: **If a conditional branch evaluates to true, the branch transfers control to the instruction at the label, otherwise we execute the next instruction.**

bne **branch if not equal**

Operation:	if (rA != rB) then PC ← PC + 4 + σ (IMM16) else PC ← PC + 4
Assembler Syntax:	`bne rA, rB, label`
Example:	`bne r6, r7, top_of_loop`
Description:	If rA != rB, then `bne` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `bne`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions:	Misaligned destination address
Instruction Type:	I
Instruction Fields:	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

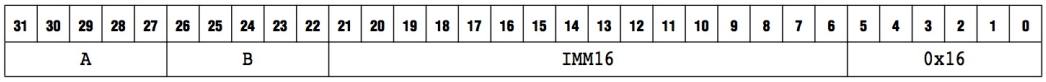
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																0x1e					

4.2.2 blt

Next we consider the *branch if less than* instruction, `blt`. While the assembly language supports all of the different conditional branches, some of them are implemented as pseudo instructions. For example, the instruction `bge r4, r5, bar` should branch to label `foo` if `r4` is less than `r5`. This can be rewritten as `blt r5, r4, bar`.

blt **branch if less than signed**

Operation:	if ((signed) rA < (signed) rB) then PC ← PC + 4 + σ (IMM16) else PC ← PC + 4
Assembler Syntax:	`blt rA, rB, label`
Example:	`blt r6, r7, top_of_loop`
Description:	If (signed) rA < (signed) rB, then `blt` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `blt`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions:	Misaligned destination address
Instruction Type:	I
Instruction Fields:	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value



4.3 If Statements

We are now ready to see how to implement conditional statements in higher-level languages. In C/C++/Java, we are permitted to write a single `if` statement, an `if/else` statement, an `if/else-if` statement, or an `if/else-if/else` statement. Let's look at how these can be expressed in assembly language.

Consider the following C/C++ fragment:

```
int x = 1;
int y = 2;
if (x < y) {
  x = x + y;
} else {
  x = x - y;
}
...
```

To re-write this in assembly language, it might look like this, assuming that we place `x` in `r2` and `y` in `r3`:

```
        movi r2, 1
        movi r3, 2
        bge  r2, r3, else
        add  r2, r2, r3
else:   sub  r2, r2, r3
        ...
```

This seems nice, but there is a problem with it. Can you find it?

Let's first look at the case when `r2` is less than or equal to `r3`. We work through the `movi` statements and reach the branch. The branch condition is true, so we jump to the else label and perform the subtraction. Sounds good.

What happens if the branch is false? The program counter is incremented to the address of the next instruction and we fall through. As in the C/C++ code, we perform the addition. Then what? We keep marching on as we always do and then perform the subtraction! Not cool.

To fix this, we need to amend our earlier solution like this:

```
            movi r2, 1
            movi r3, 2
            bge  r2, r3, else
            add  r2, r2, r3
            br done
else:       sub  r2, r2, r3
done:       ...
```

Here we have added new label and an unconditional branch. This allows us to exit the `if` block when we have completed our job. Additional unconditional branches seem strange at first, but are just making the control flow explicit within an if statement.

This model can be adapted to work with any of the `if/else` variants. We use conditional branches to take or ignore any of the `if` paths, then end each alternative with an unconditional branch to the end of the `if/else` block.

5 Loops

Loops are just a natural extension of our work that we did with conditional and unconditional branches in the previous section. Let's just look at some different loop structures and see how they look in assembly language.

5.1 while loops

Let's start with a basic while loop in C/C++:

```
while (y < 10)
  y = y + x;
```

For simplicity's sake, let's assume that `x = r2` and `y = r3`. There is one major trick to writing assembly for loops and that is that the "default" behavior is to fall-through and execute the loop. So, that means our branch must jump to the end of the loop only if the *loop condition is false*.

This runs counter to the behavior of the conditional branches in NIOS assembly. To get around this, we typically need to define our loop conditional using the *opposite sense of the high-level loop condition*. In this example, we need to branch when `y` is greater than or equal to `10`. If `y` is less than `10`, we should just fall through and keep executing the loop body.

Next question. How do we branch if `y` is greater than or equal to `10`? The conditional branch instructions take two registers as operands, not an immediate value. Working around this means that we have to stick an immediate value (10) into a register just for the branch. When we put all of these things together, we get the following:

```
        movi r4, 10
test:   bge  r3, r4, done
        add  r3, r3, r2
        br test
done:   ...
```

5.2 do-while loops

Assuming the same register assignment for `x` and `y` as in the previous example, we can make a few changes and see how we can implement a `do/while` loop in assembly. Assume that we have the following loop:

```
do {
  y = y + x;
} while (y < 10)
```

With the do/while loop, things have changed around a bit. When we get to the condition test, we need to branch back to the top of the loop if it's true and exit the loop if it's false. Rather than inverting the loop condition as we did in the basic `while` case, the loop condition and the branch condition are the same in this case. The constant in the loop condition must still be placed into a register, as before. We can implement this with the following assembly fragment:

```
        movi r4, 10
loop:   add  r3, r3, r2
        bge  r3, r4, loop
        ...
```

5.3 for loops

The last loop style that we will look at is the for loop. The easiest way to envision the assembly for the for loop is to translate it into a while loop and work from there. If we have the following for loop:

```
int i;
for (i=0; i < 10; i++)
  y = x + i;
```

This can be translated into an equivalent `while` format:

```
i = 0;
while (i < 10) {
  y = x + i;
  i++;
}
```

And then into assembly, assuming that `r2 = x`, `r3 = y`, and `r5 = i`:

```
          mov  r5, r0         # i = 0
          movi r4, 10         # constant for branch
test:     bge  r5, r4, done   # i < 10 ?
          add  r3, r2, r5     # y = x + i
          addi r5, r5, 1      # i = i + 1
          br test             # repeat
done:     ...
```

This is largely correct, but can cause unexpected results in specific cases. The original loop code defined the variable `i` outside of the loop. It should have a well-defined value at the conclusion of the loop. In this case, we expect that `i` has the value of 10 after the loop completes.

What if we change the for loop condition from `i < 10` with a condition using largest possible integer (i.e. $2^{32} - 1$ on the NIOS)? Let's call it MAXINT since this problem occurs on all computers, regardless of the actual value. The amended for loop now looks like this:

```
int i;
for (i=0; i <= MAXINT; i++)
  y = x + i;
```

If we have this loop, the conditional branch needs to be changed to a `bgt` instruction in order to accommodate the <= condition. We expect that the value of `i` at the end of the loop is equal to MAXINT, but that's not what happens with our assembly version.

When `i` is MAXINT, the `bgt` branch is false and we fall-through into the body of the loop. The `y = x + i` addition is performed and then `i` is incremented. This is where the problem occurs. Adding 1 to MAXINT causes an overflow and a branch back to the loop test. This is an error. We can fix this by restructuring our loop like this:

```
          ...
          movi r4, limit      # limit = MAXINT
          bgt  r5, r4, done   # i <= MAXINT ?

test:     add  r3, r2, r5     # y = x + i
          beq  r5, r4, done   # exit loop if i == limit (MAXINT)

          addi r5, r5, 1      # i = i + 1
          br test             # repeat
done:     ...
```

6 Arrays

Arrays are a fundamental data structure used in most high-level programming languages. You may have noticed that we haven't seen any assembly instructions that allow us to work with array data. Like so many things in assembly language, we have all of the tools needed to deal with arrays, but we'll have to do most of the work ourselves.

To start with, let's limit ourselves to dealing with one-dimensional arrays. Before we can reference an array or update values in an array, we need to know how to construct an array. The easiest way to do this is by defining an array statically, at the time we are writing our program. This is the way we will create arrays in this class. If we wish to create an array dynamically, say with the `new` operator in C or Java, or `malloc()` in C, we need to have more well-defined support for stack and heap data structures in assembly. We'll look at the stack in depth later, when we look at functions.

6.1 Static Allocation

In NIOS assembly, we can use assembler directives to reserve memory for static data. These can be used to store arrays, constants, literal strings (i.e. "hello, world!") etc. All static data must be allocated in the *data segment* of the program. These definitions must begin with the reserved keyword `.data` on a line by itself.

Inside the data segment, we can reserve a region of memory with a `.skip <size>` directive. This creates a space for `<size>` bytes and is left uninitialized. If we'd rather specify initial values, we can use `.word` or `.byte` directives to reserve both space and set initial values, one element at a time. A label is usually defined in conjunction with static data, in order to be able to reference it later.

For example, we can allocate an uninitialized array of 10 elements (i.e. `int a[10];`) as follows:

```
        .data
        .skip 40
```

We need to specify 40 bytes, because each integer is 4 bytes ($4 \times 10 = 40$ bytes). On the other hand, if we want to declare an array named `arr` initialized with the first five even numbers we can do this:

```
        .data
arr:    .word 2
        .word 4
        .word 6
        .word 8
        .word 10
```

To access an array element, we need to compute the address of the specific element that we want. The general form for accessing an array element `i` of an array `a` is:

```
a[i] = &a + size * i
```

So, if we want to access `a[3]` and the integer array `a` begins at memory address 1000, then the address of this element would be $1000 + (4 \times 3)$ or 1012. Each array element is four bytes long, so `a[0] == 1000`, `a[1] == 1004`, `a[2] == 1008`, and `a[3] == 1012`. In NIOS assembly, if the array were defined as `arr` above, the code would look like this:

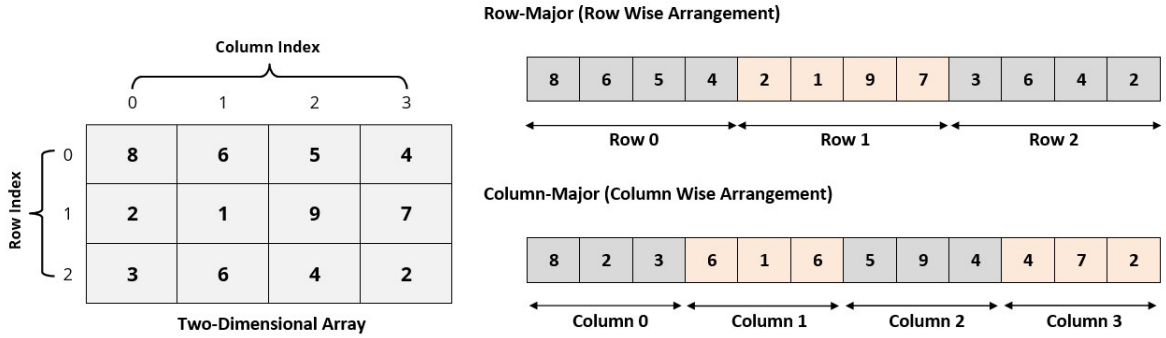
```
movi  r2, 3         #  r2 = i = 3
movia r3, arr       #  r3 = &arr
muli  r4, r2, 4     #  r4 = i * 4
add   r4, r3, r4    #  r4 = &arr + i * 4
ldw   r5, 0(r4)     #  r5 = a[3]
```

Arrays are data structures that exist only in main memory. A register is not large enough to hold more than a single word, so any contiguous array must be located in memory. To fetch the value of any array element in memory, first the address of the element must be computed using the base address of the array (`&arr`, in the example above) and adding the offset into the array (the index multiplied by the size of each element). Now we see why many programming languages start array data with index 0. `a[0] = &a + 4 * 0`, so starting arrays at zero makes the address calculation simpler.

6.2 Multi-dimensional Arrays

To support multi-dimensional arrays, we can restructure them into one-dimensional arrays. This process is called *linearization*. For example, let's take a look at two dimensional arrays to understand how this is done. The same technique can be extended to arrays of higher dimensionality.

We can represent a two-dimensional array as a one dimensional array in one of two ways. The first, *row-major order*, lays out each row in the matrix one row after the next. The alternative, *column-major order*, places each column in the matrix, one after the other. In practice, C/C++ store multi-dimensional data in row-major order, whereas a language like Fortran uses column-major. We can see the difference between each linearization in the following example:



To compute the address of an element in a multi-dimensional array, we need to know whether it is stored whether it is stored in row- or column- major order, the indices of the element we want, and the size of each dimension of an array. To compute the address of an element `a[i][j]` in an array, we would perform the following computation, if `a[]` were stored in row-major order:

```
a[i][j] = &a + (i * size * ncols) + j * size
```

In this expression `size` is the size in bytes of each element of the array and `ncols` is the number of columns in the array. In our example above, let's assume that we wish to compute the address of `a[1][2]`, which is 9. Since this is a 3×4 matrix, `ncols` is 4 and let's assume that the size of an integer is 4 bytes. Lastly, let's assume that the address of the first element of the array `&a[0][0]` is at 1000. Then we get this:

```
&a[1][2] =  &a  + (1 * size * ncols) + 2 * size
&a[1][2] = 1000 + (1 * size * ncols) + 2 * size
&a[1][2] = 1000 + (1 * 4 * ncols) + 2 * 4
&a[1][2] = 1000 + (1 * 4 * 4) + 2 * 4
&a[1][2] = 1000 + 16 + 8
&a[1][2] = 1024
```

This makes sense because we can see that an entire row of data is 4 integer elements or 16 bytes. The element we are looking for is in the second row, so we must skip over the 16 bytes of the first row. Next, we need to figure out where `a[1][2]` is in the second row. It is the third element in, so we must skip over the first two elements (i.e. 8 bytes). Putting this all together means that we must skip 24 bytes past the beginning of the array.

7 Functions

Almost all high-level programming languages support functions or subroutines. These tools are used to provide functional abstraction, reduce duplicate code, and improves re-use within software projects. Let's start off defining some terms that we'll need as we begin to explore implementing functions and subroutines in assembly language.

7.1 Terms

- **function** – a block of code that may be executed repeatedly with inputs provided as parameters. A function must always return a value. We are going to assume that functions only return a single value in assembly language. Returning multiple values can be done by storing them together in memory and returning the address from the function.
- **subroutine/procedure** – a block of code that may be executed repeatedly with inputs provided as parameters. Subroutines do not return a value.
- **caller** – the block of code that is calling a function, usually another function. In a C/C++ program, if the function `f()` is called from `main()`, then `main()` is the caller.
- **callee** – the function that is being called by the caller. In a C/C++ program, if the function `f()` is called from `main()`, then `f()` is the callee.
- **parameters** – information that is passed by the caller to the callee.
- **return value** – the value returned by the callee to the caller.
- **linkage** – the protocol for passing parameters from the caller to the callee and back again. The linkage is also referred to as the *calling conventions*. These are typically specified by the hardware manufacturer to allow programs written in different languages work together.

7.2 Calling Functions

There are two major challenges to implementing functions in assembly language. The first is to figure out how to transfer control from the function and back again. The second is how to transfer data into and out of functions. We'll start by focusing on the first problem.

To define a function, we simply create a label at the beginning of the code to be included in the function. All NIOS programs must have a special label, `_start:` defined at the beginning of the program to denote where the program should start execution. This is similar to the `main()` function in C/C++.

We might think that we can just call a function by using an unconditional branch instruction. Indeed we could jump to the function by using a `br` to the label of the function. The question becomes how do we get back to the place we were called from? The function must have another unconditional branch at the end to "return" from the function. This isn't a problem if the function is only called from one place and it can always branch back to the place it was called from.

The real problem becomes apparent when we have a function that is called from multiple locations. Let's say that a function `foo()` is called by both `a()` and `b()`. Getting to `foo()` from either `a()` or `b()` is easy, both use `br foo` to change the control flow to `foo()`. When `foo()` completes, it must return but it has no idea to branch to a location in `a()` or in `b()`! We need some additional help to remember where we came from.

7.2.1 call

To call a function from NIOS assembly language, we can use a new instruction, the `call` instruction. As can be seen from the RTN on the reference sheet, the call instruction modifies the PC in much the same way that an unconditional branch does, albeit with a larger immediate value. Perhaps more interesting is this line: `ra` $\leftarrow PC+4$. Functions require the use of a new reserved register, the *return address* register (`r31`, or just `ra`). The `call` instruction preserves the PC + 4 value (i.e. the address of the next instruction) in the return address register. By saving the address of the next instruction in the caller, the callee will know where to return to when it is done.

call **call subroutine**

Operation:	`ra` ← PC + 4 PC ← ($PC_{31..28}$: IMM26 × 4)
Assembler Syntax:	`call label`
Example:	`call write_char`
Description:	Saves the address of the next instruction in register `ra`, and transfers execution to the instruction at address ($PC_{31..28}$: IMM26 × 4).
Usage:	`call` can transfer execution anywhere within the 256-MB range determined by $PC_{31..28}$. The Nios II GNU linker does not automatically handle cases in which the address is out of this range.
Exceptions:	None
Instruction Type:	J
Instruction Fields:	`IMM26` = 26-bit unsigned immediate value

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																										0					

7.2.2 ret

Now let's look at the `ret` instruction. This instruction is used to return from a function and transfer control back to the caller. The RTN from the reference sheet is straightforward. We can see that the return instruction simply places the contents of the return address register back into the PC. This transfers control back to the instruction that follows the original `call` instruction that got us to the function.

ret **return from subroutine**

Operation:	PC ← `ra`
Assembler Syntax:	`ret`
Example:	`ret`
Description:	Transfers execution to the address in `ra`.
Usage:	Any subroutine called by `call` or `callr` must use `ret` to return.
Exceptions:	Misaligned destination address
Instruction Type:	R
Instruction Fields:	None

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
0x1f					0					0					0x05						0					0x3a					

So, problem solved, right Let's see what happens in the case where we have three functions, `a()`, `b()`, and `c()`. Let's assume that `a()` calls `b()`, and `b()` calls `c()` and the instructions have the following addresses:

```
1000   a:    add  r8, r9, r10
1004         call b
1008         addi r8, r8, 1
...
2000   b:    mul  r11, r12, r13
2004         call c
2008         subi r11, r11, 1
2012         ret
...
3000   c:    and r14, 15, 16
3004         ret
```

The program starts execution in `a()` and gets to the `call` instruction at address 1004. The call stores the $PC + 4$ into the `ra` register (1008) and updates the PC to the address of the target branch: 2000. The program transfers control to `b()`, which does the multiplication and then calls `c()`. The call to `c()` When this happens, puts the $PC + 4$ into the `ra` register (2008) and sets the PC to 3000. The program then jumps to `c()`, performs the `and` and calls `ret`. This resets the PC back to 2008 and results in us returning back to `b()`.

There is a problem, however. When we issued the second `call` instruction, we overwrote the `ra` register with 2008. What happened to the original value that `a()` put into the `ra` register (1008)? It has been lost! When we get to the `ret` statement at address 2012, it simply sets the PC to the value in the `ra` register, which is 2008 – the value set by `call c`. We will loop indefinitely here, never able to return to `a()`, as we have lost our way back.

In order to avoid this problem, we will have to keep a copy of the return address register if our current function calls another function. This copy will need to be placed into the `ra` register before we return from our function. We will keep the copy of our return address in memory, along with some other important data.

7.2.3 Passing Parameters

There are two approaches to passing parameters to functions. The caller can place parameters in registers before the `call` and the callee can get them out upon arrival in the function. In the same way we can run into trouble with nested functions and the `ra` register, we can also inadvertently overwrite our parameters if we aren't careful. Alternatively, we can pass parameters in main memory. This is safer and more flexible, but also slower.

7.3 Calling Conventions

We can see now that nested functions are going to cause a number of problems for us. The primary challenge is that there are only 32 registers that we can use, but a program may have nested function calls many levels deep. Recursive functions all consist of the same instructions but must keep track of the different parameters and local variables at each level of recursion!

To keep track of parameters, local variables, return values, and return addresses, we will need frequently store them in memory. We organize our memory using an abstraction called the *stack*. The stack consists of a collection of *stack frames*. One stack frame contains all of register values that may be overwritten by other function calls, as well as any data that can't otherwise fit in register.

The stack starts at the highest address in main memory and grows down, with one stack frame *pushed* onto the stack for each function call. Managing all of the different registers and maintaining the stack can be complicated. To make this more consistent, Altera/Intel define a set of *calling conventions* that restrict the ways we can use registers and govern how the stack is setup. These common rules ensure that software written in different languages or with different compilers are all able to interoperate.

7.3.1 Register Conventions

From here on out, we will restrict the registers we use to follow the NIOS calling conventions. The register are roles are given in the following table:

Table 7–2. Nios II ABI Register Usage (Part 1 of 2)

Register	Name	Used by Compiler	Callee Saved *(1)*	Normal Usage
r0	zero	✓		0x00000000
r1	at			Assembler temporary
r2		✓		Return value (least-significant 32 bits)
r3		✓		Return value (most-significant 32 bits)
r4		✓		Register arguments (first 32 bits)
r5		✓		Register arguments (second 32 bits)
r6		✓		Register arguments (third 32 bits)
r7		✓		Register arguments (fourth 32 bits)
r8		✓		Caller-saved general-purpose registers
r9		✓		
r10		✓		
r11		✓		
r12		✓		
r13		✓		
r14		✓		
r15		✓		
r16		✓	✓	Callee-saved general-purpose registers
r17		✓	✓	
r18		✓	✓	
r19		✓	✓	
r20		✓	✓	
r21		✓	✓	
r22		✓	*(2)*	
r23		✓	*(3)*	
r24	et			Exception temporary

Register	Name	Used by Compiler	Callee Saved *(1)*	Normal Usage
r25	bt			Break temporary
r26	gp	✓		Global pointer
r27	sp	✓		Stack pointer
r28	fp	✓	*(4)*	Frame pointer
r29	ea			Exception return address
r30	ba			■ Normal register set: Break return address ■ Shadow register sets: SSTATUS register
r31	ra	✓		Return address

The 32 registers are divided up and given specific roles that must be followed when writing NIOS assembly language programs that have functions. Our "free choice" registers are now limited to registers `r8` through `r23`. The rest of the registers are reserved for specific uses and we must use these registers according to the calling conventions.

Of the registers listed above, we will use several of the reserved registers to manage the stack and function calls:

- **stack pointer** – `sp` (`r27`) – This register maintains the address of the bottom of the stack. When we call a new function, we need to allocate a stack frame by moving the stack pointer to a lower address (*pushing* a new frame). Right before we return from a function, we return the stack pointer to the previous value (*popping* a stack frame).

  All data stored on the stack is addressed through the stack pointer register. Memory locations on the stack are referred to by their displacement from the stack pointer value. We will see how the stack is organized shortly.

- **return address** – `ra (r31)` – This register holds the address of the instruction after the `call` instruction in the caller. Nothing has changed since we talked about the return address above.
- **frame pointer** – `fp (r28)` – We won't be using this register in our programs, but compilers frequently rely on the frame pointer register to record the previous stack pointer. If the stack pointer refers to the bottom of the stack, the frame pointer refers to the top of the lowest frame on the stack.
- **return value** – `r2` – When a:t function returns a value, it is always placed in register `r2`. The callee puts the return value in `r2` and the caller can trust that the value returned by the function is in `r2` when it returns. The `r3` register can also be used for this purpose (for example, to return 64-bit values), but we will limit ourselves to only using `r2`.
- **function parameters** – `r4, r5, r6, r7` – These registers are reserved for passing the first four parameters to a function. If a function takes more than four parameters, or a parameter is larger than a 32-bit word, they must be passed in memory on the stack. These may be used within nested function calls, so the caller of a function may place its parameters in these registers, but the callee may also use them to pass parameters of a nested call. I.e. it's not safe to assume that our parameter values in these registers will still be there upon the return of the callee.
- **caller-save registers** – `r8-r15` – These registers are our general-purpose use within a function body. They may be used for local variables or for temporary values when computing complex expressions. They are called caller-save registers because it is the responsibility of the caller to save any register values that it needs to exist after a function call.
- **callee-save registers** – `r16-r23` – These registers are also usable for local variables or temporary values within a function body. They are called callee-save registers because it is the responsibility of a callee function to save them before using them and to restore the original values prior to returning to the caller.

7.3.2 Convention Sequences

To make things more complicated, any given function may act as both a caller and a callee. The only function in our NIOS program that can never be a callee is the `_start` (or `main()`) function. Any function that never calls another is called a *leaf function* and will never be a caller.

Since there are a number of rules and responsibilities for both the caller and callee, we define specific regions within the code of a function to observe the calling conventions.

A typical NIOS function starts with a label that is the same name as the function. The function begins with the *startup sequence* and ends with the *cleanup sequence*. If the function calls another function, there is a block of instructions before the call (the *prologue*) and another after (the *epilogue*). There may be an arbitrary sequence of instructions in between these sections. This is shown in Figure 3 (on the next page).

The primary concern is that we may place values in registers in section **A** of the function, call `bar()`, then use the values from **A** in section **B**. The `bar()` function may need to re-use those registers, or may call a function that does. The obvious solution is to store important data onto the stack, but the question is who is responsible for storing and restoring the data? The answer is *it depends on the register*. The function should only be using registers `r8` – `r23` for local and temporary values.

7.3.3 The 5 Rules of Calling Convention Happiness

To determine what goes where in a Nios function block, we can use five basic rules of thumb:

1. **Caller-save Registers (`r8-r15`):** If `foo()` uses any caller-save register in section **A** and needs the value in section **B** after calling `bar()`, then it must save the register on the stack.

   The register must be saved on the stack in the prologue and restored in the epilogue. Any unused caller-save registers (or used registers that are unused after the call to `bar()`) do not need to be preserved on the stack.

2. **Callee-save Registers (`r16-r23`):** If `foo()` uses any callee-save register in the function, it must be saved on the stack.

   The register must be saved on the stack in the startup sequence and restored in the cleanup sequence. Any unused callee-saved registers can be ignored. The `main()` function can ignore any callee-save save/restore requirements.

3. **Return Address (`ra`):** If `foo()` calls any function and is not the `main()` function, it must save and restore the return-address register.

   The register must be saved on the stack in the startup sequence and restored in the cleanup sequence. Functions which do not call other functions do not need to worry about saving the `ra` register.

4. **Stack Pointer (`sp`):** If any registers need to be saved as a result of rules 1, 2, or 3, then space must be reserved on the stack to save them.

   The stack must be decremented in the startup sequence enough to store each 4-byte (32-bit) value saved. E.g. If three registers are to be saved, the stack pointer (`sp`) should be decremented by 12 bytes (3×4). Decrementing the stack pointer is the first statement in the startup sequence. Incrementing the stack pointer is always the last statement in the cleanup sequence prior to the `ret` instruction.

   The exact placement of a saved register on the stack doesn't matter to us, just that it is saved and restored from the same location. For example, if we have to save `ra, r8,` and `r16`, they would need to go in the locations: `0(sp), 4(sp),` and `8(sp)` in any order, as long as you are consistent (e.g. if `r8` is saved in `4(sp)`, then it should be restored from `4(sp)`.

5. **Parameters/Return Values (`r4-r7, r2`):** If a function is called with parameters, the actual parameters must be passed in registers `r4-r7`. Similarly, any function that returns a value must return the value in `r2`. The caller copies the actual parameters in the prologue prior to the call. The caller copies the return value into a working register in the epilogue after the call.

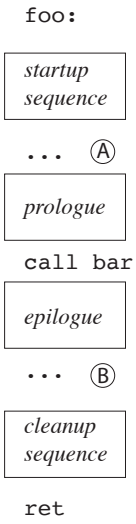


Figure 3: Typical Nios Function: `foo()`

Each of the blocks in Figure 3 are responsible for several different tasks involved in handling function calls, including managing the stack, return addresses, parameters, return values, and the saving and restoration of local and temporary registers.

1. **Startup Sequence:**

   (a) Allocate the stack frame. This is done by *subtracting* from the sp register. (because stacks grow downwards). The stack must have enough storage for a) the return address b) any caller-save registers needed (should we call another function (e.g. `bar()`) c) any callee-save registers used in `foo()`. If we don't need to save one/any of these, then you don't need to reserve room on the stack.

   (b) Save the return address register, `ra`, only if we call another function (e.g. `bar()`).

   (c) Save all callee-saved registers that we use within `foo()`. If we don't use a callee-save register (`r16-r23`), then we don't do anything

2. **Prologue:**

   (a) If the function to be called takes arguments, place all of the parameters in the parameter registers (`r4-r7`).

   (b) Save all caller-save registers to the stack.

3. **Epilogue:**

   (a) Restore all caller-save registers from the stack.

   (b) If the function returns a value, it will be in the *return value* register, `r2`.

4. **Cleanup Sequence:**

   (a) If the function returns a value, place it in the *return value* register, `r2`.

   (b) Restore all callee-save registers from the stack.

   (c) Restore the return address register from the stack.

   (d) De-allocate the stack by adding to the sp register. This should be the same amount that was subtracted in the startup sequence.

7.3.4 Saving and Restoring

To save a value to the stack, we must first ensure that space exists on the stack (see rule 4). If we have allocated enough room on the stack, then each saved value needs a unique location on the stack. For example, let's assume that we need to save the return address register, `ra`. In addition to the return address, we also must save three other registers.

We need to ensure that the stack pointer created enough space for all four (`ra` + three others) registers:

```
subi sp, sp, 16
```

Once we have that taken care of (in the startup sequence), we simply pick a free spot in the stack to place the return address. Since it's the first register to save, we'll use the first spot on the stack (zero bytes + sp):

```
stw ra, 0(sp)
```

Now the return address has been copied safely to the stack. Later, in the cleanup sequence, we need to restore it to its rightful place:

```
ldw ra, 0(sp)
```

The next saved register can be placed in 4 + sp, and the instructions to save and restore it follow rules 1-3, above.

Don't forget to de-allocate the stack frame just before returning from the function:

```
addi sp, sp, 16
```

7.4 An Example

All of these rules can be tricky to put together in one's head. Let's work through an example program that has some non-trivial functions.

```
1
2  void main(void) {
3    int x, y, z;
4    x = 4;
5    y = 1 + x;
6    z = f(x,y);
7    x = x + y + z;
8  }
9
10 int f(int a, int b) {
11   int i, j, k;
12
13   i = 3 + a;
14   j = 4 + b;
15   k = g(i+j);
16   return i + j + k;
17 }
18
19 int g(int c) {
20   int x = 2 * c;
21   return x + 2;
22 }
```

Listing 1: A small C program with functions

As shown in Listing 1, we have a small C program with a `main()` and two functions, `f()` and `g()`. We can see by inspecting the program that `main()` will be a caller (of `f()` on line 6), `f()` is both a callee (from `main()` and a caller (of `g()` on line 15), and lastly that `g()` is a leaf function that only acts as a callee.

The C syntax helps us express the program in a way that we can easily understand, but C variables don't exist in assembly language. Let's rewrite our example program, replacing the variables with register values. This gives us the program in Listing 2.

```
1
2  void main(void) {
3    int r8, r16, r2;
4    r8 = 4;
5    r16 = 1 + r8;
6    r2 = f(r8,r16);
7    r8 = r8 + r16 + r2;
8  }
9
10 int f(int r4, int r5) {
11   int r8, r16, r2;
12
13   r8 = 3 + r4;
14   r16 = 4 + r5;
15   r2 = g(r8+r16);
16   return r8 + r16 + r2;
17 }
18
19 int g(int r4) {
20   int r2 = 2 * r4;
21   return r2 + 2;
22 }
```

Listing 2: C program translated to registers

Here we can see some of the calling conventions on display. On line 6, we can see that the return value of `f()` is stored in `r2`. We can also see that the parameters to the functions are using the correct register values, `r4` and `r5` (lines 10 and 19). This helps us layout where the C variables will be stored, but we are still quite a ways away from an assembly language version.

7.4.1 First Translation

Our first pass at a translation to assembly language is shown in Listing 3. For the most part, the C expressions have been translated directly to assembly. The only part of the NIOS calling conventions that we have handled are the function parameters and return values. We can see on lines 11 and 12, that `main()` copies `x` and `y` to the parameter registers in preparation of calling `f()`. The value returned from `f()` is stored in `r2`, we use this on line 19 to compute the final value of `x`.

Inside of `f()` it is much the same, we can assume that our caller placed our parameters in registers `r4` and `r5` (lines 27,28). Prior to calling `g()`, we sum `i` and `j` and place the result into the parameter register, `r4` (line 31). After `g()` returns, we add the `i` and `j` values to the return value in `r2` (lines 37,38). The updated return value is passed back to `main()`, still in `r2`.

```
1
2  .global _start
3
4                  _start:
5                      # startup sequence
6
7                      movi r8, 4          # x is r8
8                      addi r16, r8, 1     # y is r16
9
10                     # prologue
11                     mov r4, r8          # copy x to a parameter
12                     mov r5, r16         # copy y to b parameter
13
14                     call f              # call f
15
16                     # epilogue
17
18                     add r8, r8, r16     # x = x + y
19                     add r8, r8, r2      # x = x + y + z (f(x,y))
20
21                     # cleanup sequence
22                     ret                 # return from whence we came
23
24                 f:
25                     # startup sequence
26
27                     addi r8, r4, 3      # i = 3 + a
28                     addi r16, r5, 4     # j = 4 + b
29
30                     # prologue
31                     add r4, r8, r16     # copy i + j to c parameter
32
33                     call g
34
35                     # epilogue
36
37                     add r2, r2, r8      # rval = rval (from g) + i
38                     add r2, r2, r16     # rval = rval + j
39
40                     # cleanup sequence
41                     ret                 # return from whence we came
42
43                 g:
44                    # startup sequence
45                    muli r2, r4, 2       # rval = 2 * c
46                    addi r2, r2, 2       # rval = rval + 2
47                    # cleanup sequence
48                    ret
```

Listing 3: C program translated to assembly

7.4.2 Dealing with the stack

We are now left with the hard part. We must manage the stack, return addresses, and the saving/restoring of registers. The full solution is shown in Listing 4, the numbers immediately to the right of the line numbers are meant to annotate the memory address of each instruction, not part of the assembly program.

Let's work in order of each block in the convention sequence.

main()

Startup Sequence: In `main()`, we have to allocate the stack. Since no stack has been setup, we will add a `.skip 508` directive in the `.data` section to create a 508 byte stack. We will give this region a label, let's call it `stack:` for the obvious reason. Our `main()` (and only in `main()` or `_start:`), we initialize the stack pointer, `sp`, to the top of our new memory region. This code can be found in Listing 4 on lines 6 and 66-67.

Now that we have a region of memory that we can use for our stack, let's look at the startup sequence for `main()`. As can be found on page 34 in 1(a), we need to allocate our stack frame. We do this by subtracting from the `sp` register. The question here is by how much? We don't know yet. For now, we can add a statement like `subi sp, sp, X`, where we will come back and fill in the value for X.

Looking at rule 1(b), we do call another function, so we should save the `ra` register on the stack. However, `main()` doesn't return to anywhere, so we can skip saving the return address. Last, 1(c) has to do with being a callee. Since `_start:` is never a callee, we don't have to do anything here, and our startup sequence is mostly complete.

Prologue: Here we check the rules 2(a) and 2(b) from page 34. We already have handled 2(a) by putting our parameters into the correct registers. Let's look at 2(b). Here we need to place any caller-save registers onto the stack frame that we wish to use after the call to `f()`. We can see from Listings 1 and 2, that we need the variables `x` and `y` after the function call. Our translation has `x` in `r8`, which is a caller-save register. We need to add an `stw` statement to save the value in `r8` to the stack frame. Since we haven't used the stack yet, we can put it into the first location on the stack, exactly at the location of the stack pointer: `0(sp)`. The `stw` instruction that saves `r8` can be seen on line 14. We've finished the prologue!

Recall that the stack grows downward when we allocate a new stack frame. We move the stack pointer to the bottom of the stack frame and work our way back up towards the previous frame. At this point we have accounted for all of the information that needs to be saved on the stack: our single caller-save register, `r8`. Now we know that our stack frame should be four bytes and we can go back and replace the `X` in the startup sequence with a 4, as seen on line 7.

Epilogue: The epilogue restores the state saved by the prologue. Here we simply need to restore the original value in `r8` from it's safe home on in the stack frame. The call to `f()` may have modified our original value in `r8`. Since it is a *caller-save* register, it is the caller's responsibility to restore it. Epilogue complete (line 21).

Cleanup Sequence: Looking at our rules for the cleanup sequence on page 34, we have to account for four things. First, we need to place our return value into `r2`. This technically happens on line 25, as a way of avoiding to use another temporary register. This is fine.

Next, 2(b) tells us to restore all of the callee-save registers. Since we're not a callee, we can ignore. Lastly, we de-allocate the stack frame by adding the amount we subtracted from the `sp` in the startup sequence. This was four bytes, so we add it back and we are done.

```
1
2  .global _start
3
4              _start:
5                  # startup sequence
6   996            movia sp, stack   # give stack initial value
7  1000            subi sp, sp, 4    # allocate storage for stack
8
9  1004            movi r8, 4        # x is r8
10 1008            addi r16, r8, 1   # y is r16
11
12
13                 # prologue
14 1012            stw r8, 0(sp)     # save r8 before calling f
15 1016            mov r4, r8        # copy x to a parameter
16 1020            mov r5, r16       # copy y to b parameter
17
18 1024            call f            # call f
19
20                 # epilogue
21 1028            ldw r8, 0(sp)     # restore r8 after calling f
22
23 1032            add r8, r8, r16   # x = x + y
24 1036            add r8, r8, r2    # x = x + y + z (f(x,y))
25
26                 # cleanup sequence
27 1040            addi sp, sp, 4    # deallocate storage for stack
28 1044            ret               # return from whence we came
29
30             f:
31                 # startup sequence
32 1048            subi sp, sp, 12   # allocate storage for stack frame
33 1052            stw ra, 0(sp)     # save return address
34 1056            stw r16, 4(sp)    # save caller's r16 to stack
35
36 1060            addi r8, r4, 3    # i = 3 + a
37 1064            addi r16, r5, 4   # j = 4 + b
38
39                 # prologue
40 1068            stw r8, 8(sp)     # save r8 before calling g
41 1072            add r4, r8, r16   # copy i + j to c parameter
42
43 1076            call g
44
45                 # epilogue
46 1080            ldw r8, 8(sp)     # restore r8 after calling g
47
48 1084            add r2, r2, r8    # rval = rval (from g) + i
49 1088            add r2, r2, r16   # rval = rval + j
50
51                 # cleanup sequence
52 1092            ldw r16, 4(sp)
53 1096            ldw ra, 0(sp)     # restore return address
54 1100            addi sp, sp, 12   # deallocate stack frame
55 1104            ret               # return from whence we came
56
57             g:
58               # startup sequence
59               # (nothing)
60 1108          muli r2, r4, 2      # rval = 2 * c
61 1112          addi r2, r2, 2      # rval = rval + 2
62               # cleanup sequence
63               # (nothing)
64 1116          ret
65             .skip 500
66 1616        stack:
67             .end
```

Listing 4: Final translation including memory addresses

f()

The `f()` function is both a caller and a callee, so it's the trickiest part of this example. Once again, however, we are just going to follow our rules on page 34 and go step by step:

Startup Sequence: We start knowing that we will have to allocate a stack frame, at least to save the return address since we can see that we are calling another function. Again, we can just write a line like: `subi sp, sp, X` until we know how big the stack frame will be.

Next, we save the return address to the stack, which we'll put at the bottom of the stack frame (`sp + 0`). Checking 1(c), however, gives us something new. Looking at the first draft code in Listing 3, we see that `f()` is using both `r8` and `r16`. Since `r16` is a callee-save register and `f()` is the callee of `main()`, we need to save `r16` to the stack. Because we are our own boss, we'll put it at `4 + sp`. You can see this on line 36 in Listing 4. With that, we're done with the startup sequence for now.

Prologue: Again, we have to make sure that our function parameters are in the correct parameter register, which was already done in Listing 3. Rule 2(b) tells us that we have to save any caller-save registers that we need after the function call to `g()` to the stack frame. Let's just use the next empty slot, `8 + sp`. Saving the caller-save register `r8` to the stack happens on line 40.

At this point we know that we need to store three words on the stack, with each word occupying 4 bytes. We can go back to the startup sequence and replace our X with a 12.

Epilogue: The only action we need to take here is to restore the caller-save register that we saved in the prologue. We load it from the stack frame back into `r8`.

It might be tempting to skip the saving and restoring of `r8` because we can see that `g()` doesn't use `r8`. This is true, but the calling conventions are rules that must be followed even if they are occasionally inefficient. If `g()` were to be modified to use `r8` at a later date, our code will still be correct if we follow conventions, but will fail if we were being clever. We could have avoided this by using a callee-save register for `i`/`r8` and make `g()` deal with saving/restoring it.

Cleanup Sequence: The rules for the cleanup sequence start by restoring all callee-save registers. We can see that loading the old value back into `r16` from the stack frame will leave the register in the same state it was in when we were called. (line 52)

Next, we restore the return address and de-allocate the stack by adding 12 bytes back to the `sp` and return from whence we came. (lines 53-55)

g()

This function is trivially simple! Hooray! It needs no stack because it: a) does not call another function (and therefore doesn't need to save the `ra` register and b) doesn't use any caller- or callee- save registers. The `g()` function can get by entirely by using the parameter register `r4` and the return value register `r2`.

7.4.3 A Simulation

Let's take the program in Listing 4 and simulate it's operation. Hopefully, this will help make it clear why we have to through all of this trouble just for a few lousy functions.

Let's consider the state of the world before the program begins execution. Below we can see the contents of the important registers on the left and the contents of memory (i.e. the stack) on the right. The only thing interesting here is that the PC has been set to start the program at memory address 996, which corresponds to the `_start:` label. The return address is a mystery since it is undefined at this point.

r2		rv	1616	...
r4		parameter 1	1612	...
r5		parameter 2	1608	...
r8		caller-save	1604	...
r16		callee-save	1600	...
sp		stack ptr	1596	...
ra	?	return addr	1592	...
pc	996	prog counter	1588	...

Let's execute the first instruction and see what the `movia` does:

r2		rv	1616	...
r4		parameter 1	1612	...
r5		parameter 2	1608	...
r8		caller-save	1604	...
r16		callee-save	1600	...
sp	1616	stack ptr	1596	...
ra	?	return addr	1592	...
pc	1000	prog counter	1588	...

So now we can see that the stack pointer contains the address of the `stack:` label, at memory address 1616. Our stack will grow downwards (back towards the program code!). Next, let's fast-forward a bit until we reach line 16:

r2		rv	1616	...	
r4		parameter 1	1612	4	r8
r5		parameter 2	1608	...	
r8	4	caller-save	1604	...	
r16	5	callee-save	1600	...	
sp	1612	stack ptr	1596	...	
ra	?	return addr	1592	...	
pc	1012	prog counter	1588	...	

We have saved our caller-save register, `r8` out to the stack. Let's move forward again, right when we arrive in `f()`, after the `call f` instruction on line 18.

r2		rv
r4	4	parameter 1
r5	5	parameter 2
r8	4	caller-save
r16	5	callee-save
sp	1600	stack ptr
ra	1028	return addr
pc	1048	prog counter

1616	...	
1612	4	r8
1608	...	
1604	...	
1600	...	
1596	...	
1592	...	
1588	...	

Now we see that the PC is referring to the `subi` instruction that allocates the stack frame in `f()`. The `call` instruction has placed the address of the next instruction after the call in `_start:` into the `ra` register, and `x` and `y` have been copied into the parameter registers, `r4` and `r5`.

Let's keep moving on, now to line 37, right after we have initialized `i` and `j` in `f()`:

r2		rv
r4	4	parameter 1
r5	5	parameter 2
r8	7	caller-save
r16	9	callee-save
sp	1600	stack ptr
ra	1028	return addr
pc	1064	prog counter

1616	...	
1612	4	r8
1608	...	
1604	5	r16
1600	1028	ra
1596	...	
1592	...	
1588	...	

Quite a bit has changed at this point. Upon entering `f()`, we grew the stack down by another 12 bytes and saved our return address (back to `_start:` onto the stack. We also saved `r16`, since it's a callee-save register and we are the callee. It's clear why we needed to save the registers, too. The `r8` and `r16` registers contained 4 and 5 back in `_start:`, but have been replaced with 7 and 9 in `f()`. While they are still in the parameter registers, we can also see them safe and sound on the stack. The 4 is located in `_start:`'s stack frame and the 5 is in `f()`'s frame.

If we keep moving until just after we call `g()`, we will be at the following state (line 60):

r2	34	rv
r4	16	parameter 1
r5	5	parameter 2
r8	7	caller-save
r16	9	callee-save
sp	1600	stack ptr
ra	1080	return addr
pc	1108	prog counter

1616	...	
1612	4	r8
1608	7	r8
1604	5	r16
1600	1028	ra
1596	...	
1592	...	
1588	...	

At this point we can see that the return address register has been overwritten with the address that `g()` is to return to in `f()`. We have saved the return address to get back from `f()` into `f()`'s stack frame. Also notice that the `r4` register has been overwritten with the `i+j` value from the program. If we didn't have a copy of the 4 value on the stack, it would be lost.

Next `g()` performs a multiplication, an addition, and returns to `f()`. Let's see what our state is upon returning to `f()` on line 55.

r2	50	rv
r4	16	parameter 1
r5	5	parameter 2
r8	7	caller-save
r16	5	callee-save
sp	1600	stack ptr
ra	1028	return addr
pc	1112	prog counter

1616	...	
1612	4	r8
1608	7	r8
1604	5	r16
1600	1028	ra
1596	...	
1592	...	
1588	...	

Here `g()` has finished it's computation and returned the value in `r2` to `f()`. Upon returning, `f()` has restored `r8`, even though it wasn't touched by `g()`. Next `f()` uses the returned value and computes its own return value, storing the final result in `r2` as well. On the way out the door, `f()` has restored the value in the `r16` callee-save register, so that it will where `_start:` expects it upon return. Lastly, the return address has been restored to get us back to `_start:`.

If we move back into the main function, right before termination at line 28, we get the following picture:

r2	50	rv
r4	16	parameter 1
r5	5	parameter 2
r8	59	caller-save
r16	5	callee-save
sp	1616	stack ptr
ra	1028	return addr
pc	1044	prog counter

1616	...	
1612	4	r8
1608	7	r8
1604	5	r16
1600	1028	ra
1596	...	
1592	...	
1588	...	

We can see that `r8` was restored to it's original value (4), then modified to compute the final value of the program (4 + 5 + 50). The stack frame for `f()` was de-allocated returning the `sp` to 1612 (1600 + 12). At the end of the program, `_start:` also de-allocated it's stack frame, returning the `sp` to its original value of 1616.

7.4.4 Final Thoughts

There are a couple of issues with our example program that we didn't address during the walkthrough for clarity's sake. Let's look at them now and figure out how to deal with the loose ends.

First off, should we save the `ra` from `_start:` if it has an unknown value? Well, we shouldn't. It's always a bad idea to transfer control to an unknown address. Really bad. While it's not great to start with an exception to the calling convention rules, saving `main()`'s return address to the stack is not needed.

Second, if we don't have a valid return address in our main function, we probably shouldn't call `ret` on line 28. If we remove it, though the computer simply operates as if we have a single straight-line program. It will complete the instruction on line 27, then continue on by just incrementing the PC until it reaches the first line of `f()` on line 32. This is hardly what we would expect it to do.

Normally, we would transfer control back to the operating system, but we don't have the luxury of an OS on the FPGA devices. It's conventional to simply enter into an infinite loop at the termination of our program. This is certainly not ideal, but the idiom is clear and it prevents us from repeating blocks of code unexpectedly. Here is how this might look in place of the `ret` statement in `_start:`.

```
done: br done
```

8 Instruction Encoding

To complete our understanding of how assembly language programs, we need to look at how the assembler deals with programs and connects the instructions to the hardware on the CPU. The NIOS instructions are encoded as 32-bit binary values. These 32-bits are used as inputs to the control unit on the CPU and directly control the behavior of the CPU.

How do we know how an instruction is encoded into binary? Like many hardware decisions, the specific encoding is a bit arbitary and specified by the hardware vendor. In this case, Altera/Intel provides the encoding for each instruction on the page in the instruction reference.

If we look back at the reference page for the `add` instruction we can see the following legend at the bottom of the page:

Instruction Type:	R
Instruction Fields:	A = Register index of operand rA
	B = Register index of operand rB
	C = Register index of operand rC

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					0x31						0					0x3a						

We can use this to see how any `add` instruction is represented in binary. Let's take the following instruction as an example:

```
add r3, r2, r1
```

We can see that since this is an R-type instruction, we need to encode the three register values into 5-bit fields (`r3, r2, r1`) and fill out the template as given. Recall that the RTN for the `add` instruction is: $\texttt{rC} \leftarrow \texttt{rA} + \texttt{rB}$. So, in our example `r3` is register C, `r2` is register A, and `r1` register B.

```
00010 00001 00011 11 0001 00000 11 1010
  A     B     C    3   1    0    3   a
```

Above, we followed the template, using the field-widths specified by the reference. This gives us the correct string of 0's and 1's, but isn't easily translated into hexadecimal. If we group the same string into groups of four bits, we can easily translate into hex:

```
00010 00001 00011 11 0001 00000 11 1010

0001 0000 0100 0111 1000 1000 0011 1010
  1    0    4    7    8    8    3   a
```

When we load an assembly program onto the NIOS CPU using the Altera Monitor Program, we can see the 32-bit hexadecimal value that is generated by the assembler:

```
                    _start:
                            add r3, r2, r1
                    _start:
0x00000000  1047883A        add     r3, r2, r1
```

We can use the instruction reference to generate true binary machine code (assemble), or to take a previously compiled executable program and reverse the process to read the program code in assembly-language (disassembly).

Not all instructions are encoded the same way. Let's look at a `ldw` instruction:

Instruction Type:	I
Instruction Fields:	`A` = Register index of operand rA
	`B` = Register index of operand rB
	`IMM16` = 16-bit signed immediate value

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																0x17					

Instruction format for `ldw`

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																0x37					

Instruction format for `ldwio`

Assume that we have the following assembly instruction:

```
ldw r3, -4(r2)
```

Recall that the RTN for the `ldw` instruction is: `rB` $\leftarrow$ Mem32[`rA` $+ \sigma$(IMM14)]. So, `r3` is register B, `r2` is register A, and -4 is the immediate 16-bit signed value.

First, let's compute the 16-bit 2's complement representation for -4:

```
0000 0000 0000 0100
1111 1111 1111 1011  (1's complement)
                +1
------------------
1111 1111 1111 1100
```

Now we can fill out the template as described in the reference:

```
00010 00011 1111111111111100 01 0111
  A     B        -4           1   7
```

Again, restructure this into groups of 4 bits to translate to hexadecimal:

```
00010 00011 1111111111111100 01 0111

0001 0000 1111 1111 1111 1111 0001 0111
  1    0    F    F    F    F    1    7
```

If we check the monitor program, we can verify our work:

```
                    _start:
                            ldw r3, -4(r2)
                    _start:
0x00000000  10FFFF17        ldw     r3, -4(r2)
```