Introduction to Nios® V

For Quartus® Prime 24.1

1 Introduction

This tutorial presents an introduction to the Altera® *Nios® V* processor, which is an implementation of the RISC-V processor architecture. Three versions of Nios V exist, each with different features and capabilities. Documentation for each of the three versions, designated as *compact* (Nios V/c), *microcontroller* (Nios V/m), and *general purpose* (Nios V/g), can be found by searching on the Internet for keywords such as `Nios V versions`.

This document is intended for a reader who is using a computer system with Nios V on one of the DE-series boards that are described in the `Teaching and Projects Boards` section of the FPGAcademy.org website. Thus, we focus on versions of the processor that are included in those computer systems, which are Nios V/m or Nios V/g.

Contents:

- Nios V System
- Overview of Nios V Processor Features
- Register Structure
- Accessing Memory and I/O Devices
- Addressing
- Instruction Set
- Assembler Directives
- Example Program
- Nios V Machine Timer and Software Interrupt Registers
- Exception and Interrupt Processing

2 Background

Nios V is a soft processor, defined in a hardware description language, which can be implemented in an Altera FPGA devices by using the Quartus Prime CAD system. In this tutorial we assume that the reader is using Nios V as part of a computer system that is implemented on a DE1-SoC board. If another DE-series board is used, then most of the discussion will still be applicable, but some features of the computer system might be different.

3 Nios V System

The Nios V processor can be used with a variety of other components to form a computer system. One example of such a system is shown in Figure 1. It is called the *DE1-SoC Computer with Nios V*, and is implemented on a DE1-SoC board. A complete description of this system, describing all of the peripherals connected to Nios V, is available as part of the `Computer Organization` section of the `Courses` tab on FPGAcademy.org. [1]

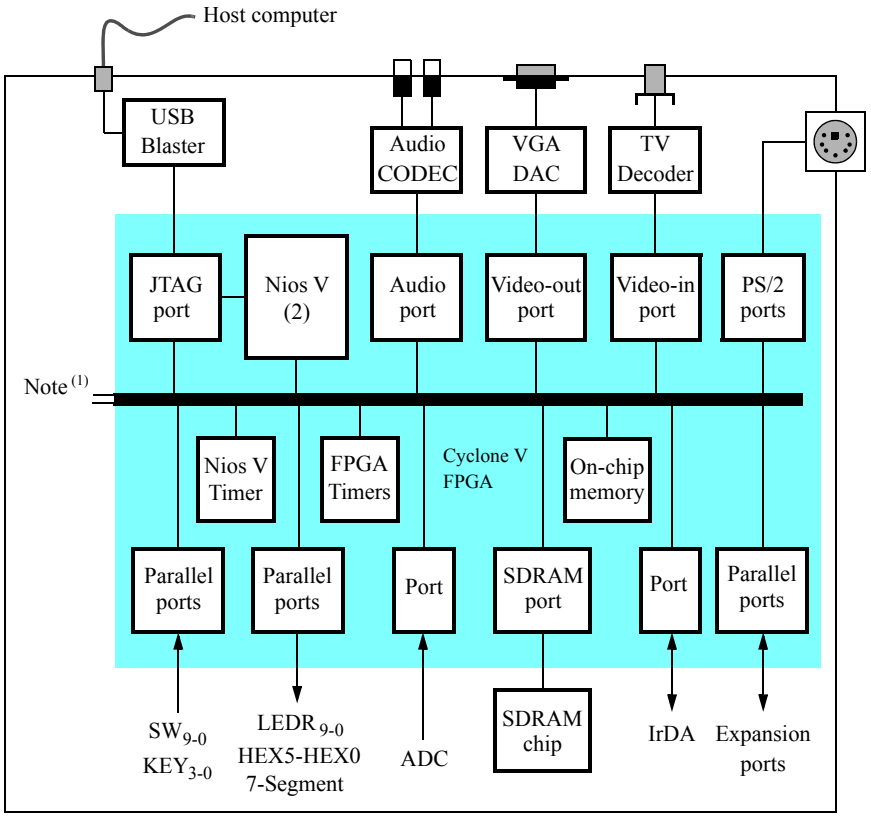

Figure 1. A Nios V system implemented on a DE1-SoC board.

[1] This computer system also includes an ARM A9 processor subsystem. It is described in the document entitled *DE1-SoC Computer System with ARM* Cortex* A9*, which is available on the FPGAcademy.org website.

4 Nios® V Processor Architecture

Nios V is based on the RISC-V processor architecture that is described in detail on the riscv.org website. Useful information for programmers using RISC-V can be found on the five-embeddev.com website. Using the terminology from the RISC-V specification, Nios V has features in the *RV32IMZicsr* architecture. This mnemonic denotes a 32-bit processor that supports integer computations (*32I*) including multiply and divide (*M*, available in Nios V/g), with special instructions that are used for control purposes (*Zicsr*).

5 Register Structure

All registers in Nios V are 32-bits long. There are 32 general-purpose registers, *x0* to *x31*, plus a program counter, *pc*. The general-purpose registers are listed in Figure 2 and are used for performing integer computations in program code.

Register *x0* is referred to as the *zero* register. It always contains the constant 0. Thus, reading this register returns the value 0, while writing to it has no effect.

Most of the other general-purpose registers can be used interchangeably in practice. However, by convention software programs usually treat the registers in a particular way, as indicated below:

- Registers *x1* is known as the *ra* register, because it is often used to hold the *return address* from a subroutine.
- Register *x2* is known as the *sp* register, because it is typically used as the *stack pointer*.
- Registers *x3* and *x4* are used as special pointer registers by compilers for high-level languages. These registers are referred to as the *global pointer* (gp) and *thread pointer* (tp), respectively.
- Registers *x5* to *x7* are known as *t0* to *t2*, because they are conventionally used to hold *temporary* data. Data in these registers is not expected to be preserved over a subroutine call/return sequence. Register *x5* is also sometimes used by compilers to hold return addresses from subroutines.
- Registers *x8* and *x9* are referred to as *s0* and *s1*, because they are conventionally used to hold data that should be *saved*, meaning that their values are expected to be preserved over a subroutine call/return sequence. Register *x8* may also used by compilers as a stack *frame pointer*.
- Registers *x10* to *x17* are known as the *a0* to *a7* registers, because they are conventionally used to hold the *arguments* for subroutines. Register *a0* is also intended to be used for the return value from a subroutine.
- Registers *x18* to *x27* are known as *s2* to *s11*. They are additional *saved* registers, like *s0* and *s1*.
- Registers *x28* to *x31* are known as *t3* to *t6*. They are additional *temporary* registers, like *t0* to *t2*.

Register	Name	Conventional Usage
x0	zero	0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6	t1	Temporary
x7	t2	Temporary
x8	s0	Saved / frame pointer
x9	s1	Saved
x10	a0	Subroutine argument / return value
x11	a1	Subroutine argument
…	…	…
x17	a7	Subroutine argument
x18	s2	Saved
x19	s3	Saved
…	…	…
x27	s11	Saved
x28	t3	Temporary
…	…	…
x31	t6	Temporary

Figure 2. General-purpose registers.

Nios V also has several 32-bit *control* registers, as indicated in Figure 3. These control registers are used to report and control the operating state of the processor. The control registers are discussed in detail in Section 8.

Register	Description
mstatus	machine status register
misa	machine ISA register
mie	machine interrupt enable register
mtvec	machine trap vector register
mepc	machine exception pc register
mcause	machine cause register
mtval	machine trap value register
mip	machine interrupt pending register

Figure 3. Control registers.

6 Accessing Memory and I/O Devices

The Nios V processor issues 32-bit addresses. The memory space is byte-addressable, and is organized in the little-endian style (the address of the least-significant byte in a word is the same as the address of the word). Memory locations can be read and written as *words* (32 bits), *halfwords* (16 bits), or *bytes* (8 bits) of data. Any input/output devices that can be accessed by the Nios V processor are memory mapped and are accessed as memory locations.

7 Nios V Instruction Set

All Nios V instructions are 32-bits long. As shown in Figure 4, these machine instructions are represented using four main types of binary encodings: *R*-type, *I*-type, *S*-type, and *U*-type. In all cases the seven bits b_{6-0} denote the *opcode*. Seven-bit fields *funct7* and three-bit fields *funct3* are used to extend the opcode and specify the operation to be executed. The instruction formats use five-bit fields for specifying general-purpose source registers (*rs1* or *rs2*) or destination registers (*rd*). Finally, the remaining bits are used to specify immediate operands, which are sign-extended by Nios V (for most instructions) to provide 32-bit operands.

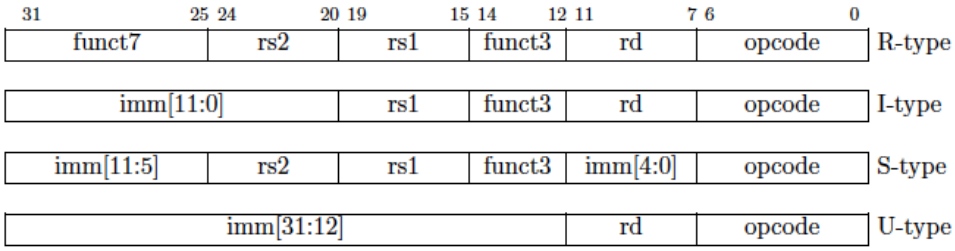

Figure 4. Formats of Nios V instructions.

In addition to machine instructions that are executed directly by the processor, there are also a number of *pseudoinstructions* that can be used in assembly-language programs. The discussion below includes several examples of such pseudoinstructions, which the Assembler implements using (one, or more) machine instructions.

The following material discusses the main features of the Nios V instruction set. More complete descriptions, including details of how each instruction is encoded, can be found online in the *RISC-V Instruction Set Manual*.

7.1 Load and Store Instructions

Load and store instructions are used to move data between memory (or I/0 interfaces) and the general-purpose registers. These are the only instructions in the processor that can directly access memory locations. All other types of instructions operate only on Nios V registers.

Load and store instructions are of I-type and S-type, respectively. Memory addresses are formed by using a *base register* and an *offset value*. For example, the load word instruction

```
lw  rd, byte_offset(rs1)
```

determines the effective address of a memory location as the sum of the value of the base register *rs1* and the *byte_offset* value. The *byte_offset* is a 12-bit constant that is sign extended to 32 bits. The data word in memory at the effective address is loaded into register *rd*. Since a word of data is being loaded, the effective address must be *word-aligned* (that is, a multiple of four).

As an example of lw, assume that register *t1* is set to the value `0x1000` [2]. Then, the instruction

```
lw  t0, 8(t1)
```

loads the 32-bit word at memory address `0x1008` into register *t0*.

The store word instruction has the format

```
sw  rs2, byte_offset(rs1)
```

It stores the value of register *rs2* into the memory location at the address computed as the sum of *byte_offset* and the value of register *rs1*.

There are load and store instructions that use operands that are only 8 or 16 bits long. They are referred to as load/store byte and load/store halfword instructions, respectively. Such load instructions are: lb (load byte), lbu (load byte unsigned), lh (load halfword), and lhu (load halfword unsigned). When a shorter operand is loaded into a 32-bit register, its value has to be adjusted to fit into the register. This is done by sign extending the 8- or 16-bit value to 32 bits in the lb and lh instructions. In the lbu and lhu instructions the operand is zero extended.

The corresponding store instructions are: sb (store byte) and sh (store halfword). The sb instruction stores the low byte of register *rs2* into the memory byte specified by the effective address. The sh instruction stores the low halfword of register *rs2*. In this case the effective address must be *halfword-aligned* (that is, a multiple of two).

When a load or store instruction does not require a *byte_offset*, it can be omitted, as in lw t0, (t1) or st t0, (t1). This shortcut is considered a pseudoinstruction by the Assembler. Other examples of short-form load and store instructions can be found in the *RISC-V Instruction Set Manual*.

7.2 Integer Arithmetic Instructions

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

7.2.1 Integer Addition and Subtraction

The instructions for adding or subtracting integer data are:

```
add  rd, rs1, rs2    (add registers)
addi rd, rs1, imm    (add immediate)
sub  rd, rs1, rs2    (subtract registers)
neg  rd, rs1         (negate register)
```

[2] Numbers can be specified in decimal (`4096`), hexadecimal (`0x1000`), or binary (`0b1000000010010110`).

The instruction

```
add  rd, rs1, rs2
```

adds the values in registers *rs1* and *rs2*, and places the sum into register *rd*. The instruction

```
addi  rd, rs1, imm
```

adds the value of register *rs1* and the sign-extended 12-bit immediate operand given in the instruction, and places the result into register *rd*. The addition operation for both add and addi is the same for both signed and unsigned operands. If it is necessary to know whether an overflow occurred (either signed or unsigned), then other instructions are needed. Overflow detection is discussed in in Section 8.2.1.

The instruction

```
sub  rd, rs1, rs2
```

subtracts the value of register *rs2* from the value of register *rs1*, and places the result into register *rd*. If carry or overflow detection is needed, then this has to be done by using additional instructions, as explained in Section 8.2.1.

There is no immediate version for subtraction, hence it is necessary to use

```
addi  rd, rs1, -imm
```

The instruction

```
neg  rd, rs1
```

is a pseudoinstruction that sets *rd* to the arithmetic negative of *rs1*. It is implemented as sub rd, x0, rs1.

7.2.2 Integer Comparison

It is sometimes necessary to compare the values in registers and to record the results. Such comparisons can be done using the instructions:

```
slt    rd, rs1, rs2     (set if less than)
sltu   rd, rs1, rs2     (set if less than unsigned)
slti   rd, rs1, imm     (set if less than immediate)
sltiu  rd, rs1, uimm    (set if less than immediate unsigned)
```

The instruction

```
slt  rd, rs1, rs2
```

sets *rd* to the value 1 if the value of register *rs1* is less than the value of *rs2*, else sets *rd* to 0.

The instruction sltu is the same as slt, except that an unsigned comparison is done.

The instruction

```
slti  rd, rs1, imm
```

sets *rd* to 1 if the value of register *rs1* is less than the sign-extended 12-bit immediate operand, else sets *rd* to 0.

The instruction sltiu is the same as slti, except that an unsigned comparison is done.

Several pseudoinstructions provide variants of the comparison instructions. Examples are:

```
seqz  rd, rs1        (set if equal to zero)
snez  rd, rs1        (set if not equal to zero)
sgtz  rd, rs1        (set if greater than zero)
sgt   rd, rs1, rs2   (set if greater than)
sgtu  rd, rs1, rs2   (set if greater than unsigned)
```

7.2.3 Integer Multiplication and Division

Multiplication and division are supported in Nios V/g. The instructions for multiplying integer data are:

```
mul     rd, rs1, rs2   (multiply)
mulh    rd, rs1, rs2   (multiply high-half)
mulhsu  rd, rs1, rs2   (multiply high-half signed/unsigned)
mulhu   rd, rs1, rs2   (multiply high-half unsigned)
```

The instruction

```
mul  rd, rs1, rs2
```

multiplies the values of registers *rs1* and *rs2*, and places the *low-order* 32 bits of the product into register *rd*. The instruction

```
mulh  rd, rs1, rs2
```

multiplies *rs1* × *rs2* and places the *high-order* 32 bits of the result into *rd*, using signed multiplication. The mulh and mul instructions together generate a 64-bit signed multiplication.

The instruction mulhsu is the same as mulh, except that the value of register *rs2* is treated as an unsigned number. The mulhsu and mul instructions together generate a 64-bit signed × unsigned multiplication.

The instruction mulhu is the same as mulh, except that the values in both registers *rs1* and *rs2* are treated as unsigned numbers. The mulhu and mul instructions together generate a 64-bit unsigned multiplication.

The instructions for generating the results of integer division are:

```
div    rd, rs1, rs2   (divide)
divu   rd, rs1, rs2   (divide unsigned)
rem    rd, rs1, rs2   (remainder)
remu   rd, rs1, rs2   (remainder unsigned)
```

The instruction

```
div  rd, rs1, rs2
```

divides the value of register *rs1* by the value of *rs2* and places the integer portion of the quotient into register *rd*. The operands are treated as signed numbers. The divu instruction is performed in the same way except that the operands are treated as unsigned numbers.

The rem instruction sets *rd* to the remainder of *rs1* ÷ *rs2*, using signed division. The remu instruction is the same as rem, except using unsigned division.

7.3 Logic Instructions

The logic instructions implement the AND, OR, exclusive OR, and NOT logical operators. They operate on data that is either in the general purpose registers or given as an immediate value. These instructions are of R-type or I-type, respectively. The logic instructions are:

```
and    rd, rs1, rs2   (AND)
andi   rd, rs1, imm   (AND immediate)
or     rd, rs1, rs2   (OR)
ori    rd, rs1, imm   (OR immediate)
xor    rd, rs1, rs2   (exclusive OR)
xori   rd, rs1, imm   (exclusive OR immediate)
not    rd, rs1        (NOT)
```

The instruction

```
and  rd, rs1, rs2
```

performs a bitwise logical AND of the values of registers *rs1* and *rs2*, and stores the result in register *rd*. Similarly, the instructions or and xor perform the bitwise logical OR and XOR operations, respectively. The instruction

```
andi  rd, rs1, imm
```

performs a bitwise logical AND of the value of register *rs1* and the 12-bit immediate operand, which is sign-extended to 32 bits, and stores the result in register *rd*. Similarly, the instructions ori and xori perform the OR and exclusive OR operations with immediate data, respectively. The instruction

```
not  rd, rs1
```

sets *rd* to the logical complement of *rs1*. It is a pseudoinstruction implemented as xori rd, rs1, -1.

7.4 Shift Instructions

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

```
sll   rd, rs1, rs2     (shift left logical)
slli  rd, rs1, uimm    (shift left logical immediate)
srl   rd, rs1, rs2     (shift right logical)
srli  rd, rs1, uimm    (shift right logical immediate)
sra   rd, rs1, rs2     (shift right arithmetic)
srai  rd, rs1, uimm    (shift right arithmetic immediate)
```

The sll instruction shifts the contents of register *rs1* to the left by the number of bit positions specified by the five least-significant bits (number in the range 0 to 31) in register *rs2*, and stores the result in register *rd*. The vacated bits on the right side of the shifted operand are filled with 0s.

The slli instruction shifts the contents of register *rs1* to the left by the number of bit positions specified by the five-bit unsigned value *uimm* given in the instruction.

The srl and srli instructions are similar to the sll and slli instructions, but they shift the value of register *rs1* to the right and fill the vacated bits on the left side with 0s.

The sra and srai instructions perform the same actions as the srl and srli instructions, except that the sign bit, $rs1_{31}$, is replicated into the vacated bits on the left side of the shifted operand.

7.5 Data Movement Instructions

A number of instructions are provided for moving data into registers. Examples are:

```
mv     rd, rs1     (move register)
lui    rd, imm     (load upper immediate)
auipc  rd, imm     (add upper immediate and pc)
li     rd, imm     (load immediate)
la     rd, label   (load address)
```

The mv instruction copies the value of register *rs1* into *rd*. It is a pseudoinstruction implemented by using addi with a zero operand. The instruction

```
lui  rd, imm
```

copies the immediate operand into the *high-order* bits of *rd*. This instruction is of U-type, which means that it supports a 20 bit *imm* operand. The lui instruction sets the lower 12 bits of *rd* to zero, which means that an instruction such as addi can be used to initialize those remaining bits. In this way, by using a combination of lui and addi, *rd* can be initialized to any 32-bit value.

The instruction

```
auipc  rd, imm
```

sets *rd* to the sum of the current value of the program counter *pc* plus *imm*. This instruction is of U-type and the *imm* value is used as an upper-immediate in the same way as for the lui instruction. This means that *imm* is padded with 12 zeros in the least-significant bit positions before being added to *pc*. The auipc instruction is not normally used directly by a programmer. Instead, it is used by the Assembler whenever it is necessary to calculate an address that is relative to the value of the program counter. The instruction

```
la  rd, label
```

sets *rd* to the memory address associated with *label*. Such a label could be associated with any statement in an assembly language program. The la pseudoinstruction is implemented as

```
auipc  rd, %high(label)      (initialize the upper 20 bits)
addi   rd, rd, %low(label)   (add an appropriate 12-bit value)
```

where %high(label) is used to initialize the 20 most-significant bits of *rd*, and %low(label) is used to add an appropriate 12-bit signed number to produce the desired 32-bit memory address. By using the auipc instruction, the calculation of a label's address works properly regardless of where the program is placed in the memory.

The instruction

```
li  rd, imm
```

sets *rd* to the value of *imm*. This pseudoinstruction can be used to initialize *rd* to any 32-bit value. In a case where *imm* fits into a 12-bit signed number, the pseudoinstruction is implemented as

```
addi  rd, x0, imm
```

But if *imm* is too large to be used as a 12-bit signed value, then li is implemented as the sequence of instructions

```
lui    rd, %high(imm)      (initialize the upper 20 bits)
addi   rd, rd, %low(imm)   (add an appropriate 12-bit value)
```

where %high(imm) is used to initialize the 20 most-significant bits of *rd*, and %low(imm) is used to add an appropriate 12-bit signed number to produce the desired 32-bit value.

7.6 Jump and Branch Instructions

The flow of execution of a program can be changed, either unconditionally or conditionally, by executing jump or branch instructions. These instructions are:

```
jal    rd, target        (jump and link)
jalr   rd, rs1, imm      (jump and link register)
beq    rs1, rs2, label   (branch if equal)
bne    rs1, rs2, label   (branch if not equal)
bge    rs1, rs2, label   (branch if greater than or equal)
bgeu   rs1, rs2, label   (branch if greater than or equal unsigned)
blt    rs1, rs2, label   (branch if less than)
bltu   rs1, rs2, label   (branch if less than unsigned)
```

The jump and link instruction

```
jal  rd, target
```

transfers execution unconditionally to the target address, which is usually a label in the program. The jal instruction also sets *rd* to the word address immediately following the jal (*pc* + 4), which means that this instruction can be used to effect a subroutine call and return sequence. In cases where the linkage register isn't needed, because the jump does not involve a subroutine, register *x0* can be used for *rd*. The jal instruction is encoded using a variant of the U-type encoding. Its 20-bit signed immediate is encoded in a particular way and is used during execution as an offset that is added to the *pc* to reach the target address. The range of the jal instruction is ±1MB.

The jalr instruction is of I-type. It forms its target address by summing the 12-bit signed *imm* value and register *rs1*. The *rd* register is set to the address of the next instruction (*pc* + 4) in the same way as for jal. By combining it with the lui instruction, jalr can jump to *any* 32-bit address: first, lui is used to load the 20 least-significant bits of the target address, and then jalr adds the remaining 12 bits. Similarly, auipc can be combined with jalr to jump to any 32-bit address relative to the *pc*.

The conditional branch instruction

```
beq  rs1, rs2, label
```

transfers execution to *label* if the value of *rs1* is equal to the value of *rs2*. If these two registers do not contain the same value, then the instruction has no effect. The other conditional branch instructions work in the same way, based on their various conditions. The conditional branches are encoded using a variant of the S-type encoding. Its 12-bit signed immediate is encoded in a particular way and is used during execution as an offset that is added to the *pc* to reach the *label*. The range of branch instructions is ±4KB.

There are also several jump and branch pseudoinstructions:

```
j     label           (jump)
jal   label           (jump and link)
ret                   (return)
beqz  rs1, label      (branch if equal to zero)
bnez  rs1, label      (branch if not equal to zero)
bgtz  rs1, label      (branch if greater than zero)
bgez  rs1, label      (branch if greater than or equal to zero)
bltz  rs1, label      (branch if less than zero)
blez  rs1, label      (branch if less than or equal to zero)
bgt   rs1, rs2, label (branch if greater than)
bgtu  rs1, rs2, label (branch if greater than unsigned)
ble   rs1, rs2, label (branch if less than or equal)
bleu  rs1, rs2, label (branch if less than or equal unsigned)
```

The j label pseudoinstruction jumps unconditionally to the given *label*. It is implemented as jal x0, label. The jal label pseudoinstruction is implemented as jal ra, label.

The ret pseudoinstruction is used to return from a subroutine to its caller. It is implemented as jalr x0, ra, 0.

The conditional branch instructions that end with z compare *rs1* to the value 0. For example, beqz branches to *label* if the value of *rs1* is equal to 0. It is implemented as beq rs1, x0, label. Similarly, the bnez, bgtz, bgez, bltz, and blez instructions provide other conditions that compare *rs1* to 0. The instructions bgt, bgtu, ble, and bleu provide additional conditions that compare *rs1* and *rs2*.

7.7 Miscellaneous Instructions

Some miscellaneous Nios V instruction that do not fit within the above categories are:

nop	(no operation)
ecall	(environment call)
mret	(machine return)
wfi	(wait for interrupt)

The nop instruction doesn't have any effect and simply advances the program counter. It is a pseudoinstruction implemented as addi x0, x0, 0.

The rest of the miscellaneous instructions listed above are used in the context of exceptions or interrupts, which are discussed in detail in Section 11.

The ecall instruction causes an exception to occur, with the exception code set to the value 3.

The mret instruction is used at the end of a trap handler routine to return to the interrupted program.

The purpose of the wfi instruction is to pause the execution of a program until an interrupt occurs. In some cases, wfi may have no effect and behave like a nop instruction.

8 Control Registers

As mentioned in Section 5, Nios V has a number of control registers that are used to report and control the state of the processor. Figure 5 shows that each register contains a number of *bit-fields*, which serve different purposes, and each register has an associated address. These control-register addresses are 12 bits wide and are not part of the normal memory address space. They are used by special instructions, described in Section 8.1, that can read and modify the control register contents.

The control registers are used as follows:

- The *mstatus* (machine status) register reports and controls the processor operating state. It has the bit fields:
  - *MPP* (machine previous privilege): these bits have the value 11 for Nios V. This setting represents *machine mode*, which is the only processor *mode* implemented in Nios V. Other RISC-V modes that may be implemented in different processors are described in the document *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, which is available on the riscv.org website.

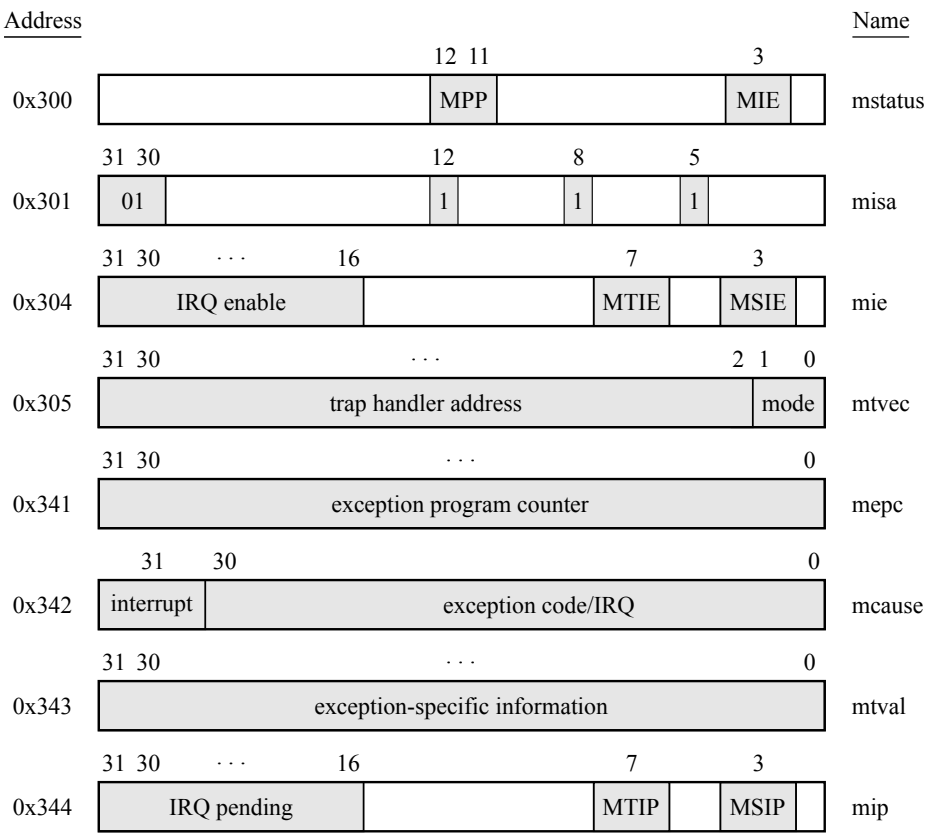

Figure 5. Nios V control register bit-fields.

  - *MIE* (machine interrupt enable): this bit allows a programmer to control interrupt handling for the Nios V processor. Interrupts are disabled if *MIE* = 0 and enabled when *MIE* = 1. Interrupts are discussed in Section 11.2.

- The *misa* (machine ISA) register reports the processor architecture. Programmers can examine the bits of this register to see which features of the RISC-V architecture have been included in Nios V. Bits b_{31-30} indicate the bit-width of the processor. For Nios V these bits have the value `01`, denoting a 32-bit processor. Also, bit $b_8 = 1$ to indicate that integer computations are supported. If multiplication and division are supported then bit b_{12} will be set to 1, and if single-precision floating point is supported then bit b_5 will be 1.

- The *mie* (machine interrupt enable) register allows individual Nios V interrupts to be enabled or disabled:

  - *MSIE* (machine software interrupt enable): disables or enables software interrupts, which are discussed in Section 11.1. Software interrupts are disabled if *MSIE* = 0 and enabled when *MSIE* = 1.
  - *MTIE* (machine timer interrupt enable): disables or enables interrupts from the built-in Nios V timer, which is discussed in Section 8.2. Timer interrupts are disabled if *MTIE* = 0 and enabled when *MTIE* = 1.
  - IRQ (interrupt request) enable: allows up to 16 external sources of hardware interrupts to be disabled or enabled. Hardware interrupts are discussed in Section 11.2. Setting each IRQ bit to 0 disables the corresponding hardware interrupt and setting it to 1 enables the interrupt.

- The *mtvec* (machine trap vector) register is used to hold the address to which the Nios V processor should jump when an exception occurs. Example code using the *mtvec* register is given in Section 11.1. There are two bit-fields:
  - *trap handler address*: a programmer stores into this field the address of a trap handler routine.
  - *mode*: a programmer sets these bits to `00` when using the basic interrupt mode, and to `01` for the vectored interrupt mode. Example code using the *mode* field is given in Section 11.3.
- The *mepc* (machine exception program counter) register holds the address of the instruction that was executing when an interrupt or other type of exception occurred. This register is used by the *mret* instruction to effect a return from a trap handler routine, as described in Section 11.1.
- The *mcause* (machine cause) register provides information that identifies the cause of an exception or interrupt. There are two bit-fields:
  - *interrupt*: when a trap occurs, this field is set to 1 if the trap was caused by an interrupt, otherwise this field is set to 0.
  - *exception code*/IRQ: this field identifies the cause of a trap. For exceptions, various possible *codes* are described in Section 11.1. For interrupts, this field gives the IRQ number of the interrupting device.
- The *mtval* (machine trap value) register may provide information that is useful for handling a trap. For example, *mtval* could provide the address that caused an alignment exception.
- The *mip* (machine interrupt pending) register indicates which interrupt(s) are pending:
  - *MSIP* (machine software interrupt pending): a programmer can set or reset this bit by writing to a memory-mapped control register, described in Section 11.1, which is used to effect a *software interrupt*.
  - *MTIP* (machine timer interrupt pending): this bit has the value 1 if an interrupt from the Nios V machine timer is pending. A programmer can reset this bit by writing to the machine timer registers, as discussed in Section 8.2.
  - IRQ (interrupt request) pending: each of these bits has the value 1 if the corresponding external interrupt is pending. A bit can be reset by following the required procedure for the interrupting device.

8.1 Control Register Instructions

The Nios V control registers can be read and written by the special instructions:

```
csrrw   rd, csr, rs1     (CSR read/write)
csrrwi  rd, csr, uimm    (CSR read/write immediate)
csrrs   rd, csr, rs1     (CSR read/set)
csrrsi  rd, csr, uimm    (CSR read/set immediate)
csrrc   rd, csr, rs1     (CSR read/clear)
csrrci  rd, csr, uimm    (CSR read/clear immediate)
```

Each of these instructions refers to a control register by its address, *csr*, as given in Figure 5. The instruction

```
csrrw  rd, csr, rs1
```

reads the value of the selected control register and copies it into *rd*. Then, the instruction writes the value of *rs1* into the control register bits (that are writable). The csrrwi instruction works in the same manner, except using an unsigned immediate operand. The instruction

```
csrrs  rd, csr, rs1
```

reads the value of the selected control register and copies it into *rd*. Then, the instruction uses the value in *rs1* as a bit-mask and sets the corresponding bits in the control register to `1`. The csrrsi instruction works in the same manner, except using an unsigned immediate operand. Analogously, the csrrc and csrrci instructions are used to clear bits selected by the bit-mask in *rs1*.

There are also several pseudoinstructions for use with control registers, which avoid the need to specify the *zero* register when it would just serve as a placeholder:

```
csrc   csr, rs1     (CSR clear)
csrci  csr, uimm    (CSR clear immediate)
csrr   rd, csr      (CSR read)
csrs   csr, rs1     (CSR set)
csrsi  csr, uimm    (CSR set immediate)
csrw   csr, rs1     (CSR write)
csrwi  csr, uimm    (CSR write immediate)
```

8.2 Nios V Machine Timer and Software Interrupt Registers

Nios V includes a 64-bit internal timer that is available to application programmers. The timer is reset to 0 when the processor is powered on, and then monotonically increases for every processor clock cycle. Since it is 64-bits wide, the timer will not overflow in any realistic amount of time. Two memory-mapped registers are associated with the timer, called *mtime* (machine time) and *mtimecmp* (machine time compare). The *mtime* register provides the current timer value, and the *mtimecmp* register can be used to cause a timer interrupt. A Nios V timer interrupt will be pending whenever the value of *mtime* reaches or exceeds the value of *mtimecmp*. Interrupts are discussed in Section 11.2.

Since they are 64-bits wide, both *mtime* and *mtimecmp* comprise two 32-bit memory-mapped registers, one for the *low* word and the other for the *high* word. Hence, two lw instructions are needed to read a value from each of these registers, and two sw instructions are needed to write a value into each of them.

Nios V also contains a memory-mapped register called *msip* (machine software interrupt pending), which can be used by an application programmer to cause a *software interrupt*. Bit b_0 in this register is reflected in the *MSIP* bit of the *mip* register, from Figure 5. A programmer can create a software interrupt by writing a 1 into bit b_0 of the *msip* register. The other 31 bits in this register are not used. Interrupts are discussed further in Section 11.2.

The *mtime*, *mtimecmp* and *msip* registers are part of a module in Nios V called the *Advanced Core Local Interrupt Specification* (ACLINT). As illustrated in Figure 6, a *base* address is assigned to the *ACLINT*, and each of the registers inside it has an address that is *offset* from this base. The value of the *ACLINT* base address is system dependent.

Figure 6. Nios V memory-mapped control registers.

8.2.1 Carry and Overflow Detection

As pointed out in Section 7.2.1, the Nios V addition and subtraction instructions perform the corresponding operations in the same way for both signed and unsigned operands. The possible carry and arithmetic overflow conditions are not detected, because Nios V does not contain condition flags that might be set as a result. These conditions can be detected by using additional instructions.

Assume that we are dealing with unsigned numbers and consider the add instruction

```
add  t2, t1, t0
```

Having executed this instruction, a possible occurrence of a carry out of the most-significant bit ($t2_{31}$) can be detected by checking whether the unsigned sum (in register *t2*) is less than one of the unsigned operands. For example, if this instruction is followed by the instruction

```
sltu t3, t2, t0
```

then the value of the carry bit will be in register *t3*.

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

```
add   t2, t1, t0
bltu  t2, t0, label
```

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

```
add  t2, t0, t1          # the required add operation
xor  t3, t2, t0          # compare signs of sum and t0
xor  t4, t2, t1          # compare signs of sum and t1
and  t3, t3, t4          # set t3[31] = 1 if ((t0[31] == t1[31]) != t2[31])
blt  t3, zero, label     # branch if overflow occurred
```

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

```
sub   t2, t1, t0
bltu  t1, t0, label
```

The arithmetic overflow in a subtract operation is detected by comparing the sign of the generated difference with the signs of the operands. Overflow occurs if the operands in registers *t1* and *t0* have different signs, and the sign of the difference in register *t2* is different than the sign of *t1*. Thus, a conditional branch based on the arithmetic overflow can be achieved as follows:

```
sub   t2, t1, t0        # the required subtract operation
xor   t3, t1, t0        # compare signs of t1 and t0
xor   t4, t1, t2        # compare signs of t1 and t2
and   t3, t3, t4        # set t3_31 = 1 if ((t1_31 != t2_31) && (t1_31 != t2_31))
blt   t3, zero, label   # branch if overflow occurred
```

9 Assembler Directives

The Nios V Assembler conforms to the widely used GNU* Assembler, which is software available in the public domain. Thus, the GNU Assembler directives can be used in Nios V programs. Assembler directives begin with a period. We describe some of the more frequently-used assembler directives in Figure 7.

Directive	Purpose
.ascii "*string*"	A string of ASCII characters is loaded into consecutive byte addresses in the memory. Multiple strings, separated by commas, can be specified.
.asciz "*string*"	The same as .ascii, except that each string is followed (terminated) by a zero byte.
.byte *expressions*	Expressions separated by commas are specified. Each expression is assembled into a byte. Examples of expressions are: 8, 5 + LABEL, and K − 6.
.data	Identifies the data that should be placed in the data section of the memory. The desired memory location for the data section is system dependent.
.end	Marks the end of the source code file; everything after this directive is ignored by the Assembler.
.equ *symbol, expression*	Sets the value of *symbol* to *expression*. This directive can also be specified as .eqv.

Figure 7. Directives (Part *a*).

Directive	Purpose
.global *symbol*	Makes *symbol* visible outside of the assembled object file. This directive can also be specified as .globl.
.hword *expressions*	Expressions separated by commas are specified. Each expression is assembled into a half-word (16-bit) number.
.include "*filename*"	Provides a mechanism for including supporting files in a source program.
.org *new-lc*	Advances the location counter to the address *new-lc*. The .org directive may only increase the location counter, or leave it unchanged; it cannot move the location counter backwards.
.skip *size*	Emits the number of bytes specified in *size*; the value of each byte is zero.
.text	Identifies the code that should be placed in the text section of the memory. The desired memory location for the text section is system specific.
.word *expressions*	Expressions separated by commas are specified. Each expression is assembled into a 32-bit value.

Figure 7. Directives (Part *b*).

10 Example Program

As an illustration of Nios V instructions and assembler directives, Figure 8 gives an assembly-language program that computes a dot product of two vectors, *A* and *B*. The vectors have *n* elements. The required computation is

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

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

The program ends by continuously looping to the *stop* label.

```
# Program that calculates a dot product of two vectors

.global _start
_start:     la      s0, Avector     # pointer to vector A
            la      s1, Bvector     # pointer to vector B
            la      t0, N           # pointer to N
            lw      s2, (t0)        # s2 = N (number of vector elements)
            li      s3, 0           # s3 will hold the final result

loop:       lw      a0, (s0)        # load next element of vector A
            lw      a1, (s1)        # load next element of vector B
            jal     mult            # compute product
            add     s3, s3, a0      # add to the sum
            addi    s0, s0, 4       # point to next element of vector A
            addi    s1, s1, 4       # point to next element of vector B
            addi    s2, s2, -1      # count down
            bnez    s2, loop

            la      t0, dot_p       # pointer to the final result
            sw      s3, (t0)        # store the result
stop:       j       stop

# Multiplies a0 x a1, using successive addition
# Returns the product in a0
mult:       mv      t0, zero        # product
            mv      t1, zero        # for checking sign of result
            bgez    a1, mloop       # is a1 a positive value?
            neg     a1, a1          # ...if not, then negate a1
            li      t1, 1           # remember to negate the product
mloop:      beqz    a1, mdone       # done adding yet?
            add     t0, t0, a0      # add the multiplier to the product
            addi    a1, a1, -1      # decrement the multiplicand
            j       mloop
mdone:      beqz    t1, mult_ret    # is sign of product correct?
            neg     t0, t0          # flip sign of result
mult_ret:   mv      a0, t0          # return result
            ret

N:          .word   6
Avector:    .word   5, 3, -6, 19, 8, 12
Bvector:    .word   2, 14, -3, 2, -5, 36
dot_p:      .word   0
```

Figure 8. A program that computes the dot product of two vectors.

In Figure 8, the directive

```
.global _start
```

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

The program includes some sample data, which illustrate how the .word directive can be used to place data items into memory.

Since the program performs multiplication by using a subroutine, it can be executed on a Nios V/m processor, which does not have the RISC-V multiplication and division extension. If the program were executed on a Nios V/g processor, which does have the multiplication and division extension, then the *mult* subroutine could be removed and replaced with a mul instruction.

11 Exceptions and Interrupts

An unexpected change in the flow of execution in Nios V programs can be caused by a *trap*, which is either an *exception* or *interrupt*. Exceptions are typically caused by error conditions, described in Section 11.1, and interrupts are caused by hardware devices or software, discussed in Section 11.2. The instruction that is executing when such an exception or interrupt occurs is not completed; instead, the instruction is stopped and Nios V transfers control to a *trap handler* routine.

11.1 Exceptions

An exception can occur as the result of an error condition, or it can be caused by executing an ecall instruction. When an exception happens Nios V performs a number of actions:

- The value of the *pc* is copied into the *mepc* register.
- Bit b_{31} of the *mcause* register is set to 0, to indicate the occurrence of an exception. Also, a unique *exception code* that identifies the specific type of exception that has occurred is placed in *mcause*. Some examples of exception codes are:
  - 0 (instruction not word aligned): this exception occurs if Nios V attempts to fetch an instruction when the value of the *pc* is not a multiple of four.
  - 2 (illegal instruction): this exception occurs if the machine code fetched from the *pc* address does not correspond to a supported Nios V instruction.
  - 4 (load address not aligned): this exception occurs if lw is executed to read from an address that is not a multiple of four, or if lh is executed to read from an address that is not a multiple of two.
  - 6 (store address not aligned): this exception occurs if sw is executed to write to an address that is not a multiple of four, or if sh is executed to write to an address that is not a multiple of two.
  - 11 (ecall): this exception occurs when the environment call instruction is executed.

  Other exception codes also exist, and are described in the RISC-V specification.

- For some exceptions a value is written into the *mtval* register. For example, if an address alignment exception happens, then the faulty address would be written by Nios V into *mtval*.
- The value of the *trap handler address* in the *mtvec* register is copied into the *pc*.

A typical trap handler routine would examine the contents of the *mcause* register to determine why the exception happened, and then take an appropriate action.

11.2 Interrupts

By default interrupts are not enabled in Nios V. To make interrupts visible to the processor, software code must performs the following two actions:

- Set the *MIE* bit in the *mstatus* register to 1.
- For each source of interrupt that should be enabled, set the corresponding enable bit to 1 in the *mie* register.

When an interrupt occurs, its corresponding bit in the *mip* register will be set to 1, to show that the interrupt is pending. If enabled, this interrupt will cause Nios V to perform a number of actions:

- The currently-executing instruction is cancelled, and the value of the *pc* is copied into the *mepc* register.
- The value of the *MIE* bit in the *mstatus* register is set to 0, preventing nested interrupts.
- Bit b_{31} of the *mcause* register is set to 1, to indicate the occurrence of an interrupt. Also, the IRQ (interrupt request) number of the interrupting device is written into the *exception code*/IRQ field of the *mcause* register. Possible IRQ numbers are:
  - 3 (software interrupt): this interrupt is pending when software writes a 1 into the memory-mapped *msip* register, from Figure 6.
  - 7 (machine timer interrupt): this interrupt is pending when the value of *mtime*, from Figure 6, is greater than or equal to the value of *mtimecmp*.
  - ≥ 16 (external interrupt): each of these interrupts is pending when the corresponding external device sends an interrupt request signal to Nios V.
- The value of the *trap handler address* is copied into the *pc*. Two schemes are supported for determining the trap address, depending on the *mode* bits in the *mtvec* register, from Figure 5. The possible mode bit values are:
  - 00 (*Basic mode*): for all interrupts, the *trap handler address* in the *mtvec* register is copied into the *pc*.
  - 01 (*Vectored mode*): the trap address depends on the interrupt's IRQ number. The address written into the *pc* is calculated as the *trap handler address* in *mtvec* plus 4×IRQ. [3]

For each interrupt, the trap handler routine typically executes specific software code that clears the source of the IRQ and performs whatever actions are needed to service it. The trap handler then returns to the interrupted program by executing the mret instruction.

[3] At this time Nios V does not support the RISC-V vectored interrupt mode. It is described here for completeness and for future use.

11.2.1 Software Interrupts

As indicated above, a software program can cause a software interrupt to become pending by writing a 1 into bit b_0 of the *msip* register, from Figure 6. This bit-value will be reflected in the *MSIP* bit in the *mip* register shown in Figure 5. If this interrupt is enabled, the normal interrupt processing actions described above will take place. The software interrupt can be cleared by writing a 0 into bit b_0 of the *msip* register.

11.3 Interrupts Processing Examples

Figure 9 gives an example of code that uses the basic Nios V trap handling mode. The code enables interrupts from the Nios V machine timer and uses them to display a binary counter on an output port that has a set of red lights (LEDs). The addresses of the timer registers, *mtime* and *mtimecmp*, as well as the red light port, called *LEDR*, are taken from the *DE1-SoC Computer System with Nios V*, shown in Figure 1.

Lines 1 and 2 in Figure 9*a* define some symbolic names that are used in the code for the I/O addresses. The symbol *clock_rate* corresponds to the frequency of the clock signal that is connected to the Nios V processor. This clock frequency is system dependent, and so the value specified in Line 3, which is 100 million, is just an example. The intent of the *clock_rate* value in this example to cause a timer interrupt every one second.

Line 5 initializes the stack pointer to an address within the 64 MB SDRAM in the DE1-SoC system, and Line 6 calls a subroutine that initializes the machine timer. Interrupts are enabled in Lines 8 to 12. First, the address of the trap handler routine is written into the *mtvec* register, and then machine timer interrupts are enabled by setting bit b_7 of the *mie* register. Finally, interrupts are enabled in Nios V by setting bit b_3 of the *mstatus* register.

```
1  .equ    LEDR_BASE, 0xff200000
2  .equ    MTIME_BASE, 0xff202100
3  .equ    clock_rate, 100000000
4  .global _start
5  _start:     li      sp, 0x40000      # initialize the stack location
6              jal     set_timer        # initialize the timer
7
8              la      t0, handler
9              csrw    mtvec, t0        # set trap address
10             li      t0, 0b10000000   # set the enable pattern
11             csrs    mie, t0          # enable timer interrupts
12             csrsi   mstatus, 0x8     # enable global interrupts
13
14             la      s0, counter      # pointer to counter
15             la      s1, LEDR_BASE    # pointer to red lights
16             sw      zero, (s1)       # turn off all lights
17 loop:       wfi
18             lw      t0, (s0)         # load the counter value
19             sw      t0, (s1)         # write to the lights
20             j       loop
```

Figure 9. An example program using basic interrupts (Part *a*).

```
21  # Trap handler
22  handler:    addi    sp, sp, -8        # save regs that will be modified
23              sw      t1, 4(sp)
24              sw      t0, (sp)
25              # check for cause of trap
26              csrr    t0, mcause        # read mcause register
27              li      t1, 0x80000007    # pattern to check interrupt bit
28                                        # and IRQ = 7
29  stay:       bne     t0, t1, stay      # unexpected cause of exception
30
31              # Restart the timer. This is one way to clear the interrupt
32              la      t0, MTIME_BASE
33              sw      zero, (t0)
34
35              la      t0, counter       # pointer to counter
36              lw      t1, (t0)          # read counter value
37              addi    t1, t1, 1         # increment the counter
38              sw      t1, (t0)          # store counter to memory
39
40              lw      t0, (sp)          # restore regs
41              lw      t1, 4(sp)
42              addi    sp, sp, 8
43              mret
44
45  # Set timeout to 1 second
46  set_timer:  la      t0, MTIME_BASE    # set address
47              sw      zero, (t0)        # reset mtime low word
48              sw      zero, 4(t0)       # reset mtime high word
49
50              li      t1, clock_rate
51              sw      t1, 8(t0)         # set mtimecmp low word
52              sw      zero, 12(t0)      # set mtimecmp high word
53              ret
54
55  counter:    .word   0                 # the counter to be displayed
```

Figure 9. An example program using basic interrupts (Part *b*).

Lines 14 to 20 in Figure 9*a* display the counter value in an endless loop, waiting for an interrupt to occur on each iteration.

The trap handler routine is given in Lines 22 to 43 of Figure 9*b*. It first saves the temporary registers that will be modified (*t0* and *t1*), and then reads the *mcause* register. To ensure that a timer interrupt has occurred, Line 27 checks that the interrupt bit, b_{31}, in *mcause* has the value 1, and the exception code field is equal to 7. If this check is not satisfied, then some kind of error has occurred and the code stops on Line 29. Otherwise, the handler resets the timer, in Line 33. This action *clears* the timer interrupt, since it makes the value of *mtime* less than the value of *mtimecmp*. Also, this action resets the timer for its next timeout.

Lines 35 to 38 of Figure 9*b* increment the counter value that is being displayed on the red lights by the main program. Finally, the last part of the trap handler restores the temporary registers that it has modified and then executes the mret instruction to return to the interrupted program.

The code that initializes the machine timer is shown at the end of Figure 9*b*. It sets *mtime* to 0 and *mtimecmp* to the value of the system clock frequency, which will result in one-second timeouts.

Instead of resetting *mtime* to zero in the trap handler, as described above, another way of handling the timer would be to increment *mtimecmp* when each interrupt occurs. This approach is more complex, but would be preferable in a system where it is not desirable to reset the *mtime* value. For this alternative approach, Figure 10 shows a version of *set_timer* that initializes *mtimecmp* to the current value of the 64-bit *mtime* register. This operation requires a loop to check for overflow from the low-to-high 32-bit parts of *mtime* that could happen during the read operation. The code then adds the *clock_rate* value to the value that was read from *mtime*, and stores the resulting sum into *mtimecmp*, accounting for the carry that could be involved in this 64-bit operation. Figure 11 shows the part of the corresponding trap handler that would need to be changed for this alternative approach of incrementing *mtimecmp*.

Figure 12 gives an example of code that uses the vectored interrupt mode (see Note 3). Two types of interrupts are used in this example: a software interrupt and timer interrupts. A key difference from the code in Figure 9 is illustrated in Lines 37 to 39, which write the handler address into *mtvec*, but with its *mode* bits set to the value 01. Also, Lines 40 and 41 set two interrupt enable bits in *mie*: bit b_7 (machine timer) and bit b_3 (software interrupt).

A software interrupt is generated by the code in Lines 45 to 47, by writing a 1 into bit b_0 of the *msip* register.

```
1  # Set timeout to 1 second
2  set_timer:  la      t0, MTIME_BASE   # set address
3              # read the current time
4  tloop:      lw      t2, 4(t0)        # read mtime high
5              lw      t1, (t0)         # read mtime low
6              lw      t3, 4(t0)        # read high again
7              bne     t3, t2, tloop    # check for overflow from low to high
8
9              # current time is t2:t1
10             li      t3, clock_rate
11             add     t3, t3, t1       # add one second to current time
12             sw      t3, 8(t0)        # write to mtimecmp low
13             sltu    t3, t3, t1       # check for carry-out
14             add     t2, t2, t3       # increment (t3 = carry-out)
15             sw      t2, 12(t0)       # write to mtimecmp high
16
17             ret
```

Figure 10. An alternative way of initializing the timer.

```
18              ... code not shown
19              # update the timer for the next interrupt cycle
20              la      t0, MTIME_BASE
21              lw      t1, 8(t0)         # read mtimecmp low
22              li      t2, clock_rate
23              add     t2, t2, t1        # add one second to mtimecmp
24              sw      t2, 8(t0)         # write to mtimecmp low
25              sltu    t2, t2, t1        # check for carry-out from addition
26
27              lw      t1, 12(t0)        # read mtimecmp high
28              add     t1, t1, t2        # increment (t2 = carry-out)
29              sw      t1, 12(t0)        # write to mtimecmp high
30              ... code not shown
```

Figure 11. Part of the trap handler for the alternative approach.

```
31  ... directives (see Figure 9) not shown
32
33  .global _start
34  _start:     li      sp, 0x40000       # initialize the stack location
35              jal     set_timer         # initialize the timer
36
37              la      t0, handler       # get handler address
38              ori     t0, t0, 0b1       # set vector mode
39              csrw    mtvec, t0         # set trap address and mode
40              li      t0, 0b10001000    # set the enable pattern
41              csrs    mie, t0           # timer & software interrupts
42              csrsi   mstatus, 0x8      # enable global interrupts
43
44              # Make a software interrupt happen
45              la      t0, MTIME_BASE    # base address
46              li      t1, 1             # pattern to write to msip
47              sw      t1, 16(t0)        # write to msip (sw interrupt)
48
49              la      s0, counter       # pointer to counter
50              la      s1, LEDR_BASE     # pointer to red lights
51  loop:       wfi
52              lw      t0, (s0)          # load the counter value
53              sw      t0, (s1)          # write to the lights
54              j       loop
```

Figure 12. An example program using vectored interrupts (Part *a*).

```
55  # Trap handler
56  handler:    j       exception
57              .org    handler + 3 * 4  # IRQ 3: software interrupt
58              j       IRQ_3
59              .org    handler + 7 * 4  # IRQ 7: timer interrupt
60              j       IRQ_7
61
62  exception:  j       exception        # not handled in this code
63
64  # software interrupt handler
65  IRQ_3:      addi    sp, sp, -8       # save regs that will be modified
66              sw      t1, 4(sp)
67              sw      t0, (sp)
68
69              la      t0, MTIME_BASE   # base address
70              sw      zero, 16(t0)     # clear software interrupt in msip
71              la      t0, counter      # pointer to the counter
72              li      t1, 0b1111111110
73              sw      t1, (t0)         # write to the counter
74
75              lw      t0, (sp)         # restore regs
76              lw      t1, 4(sp)
77              addi    sp, sp, 8
78              mret
79
80  # timer interrupt handler
81  IRQ_7:      addi    sp, sp, -8       # save regs that will be modified
82              sw      t1, 4(sp)
83              sw      t0, (sp)
84
85              # Restart the timer
86              la      t0, MTIME_BASE
87              sw      zero, (t0)       # write to mtime
88
89              ... some code not shown (see Figure 9)
90              mret
91
92  set_timer:  ... code not shown (see Figure 9)
93              ret
94
95  counter:    .word   0                # the counter to be displayed
```

Figure 12. An example program using vectored interrupts (Part *b*).

Lines 56 to 60 of Figure 12*b* provide three trap vectors, at the addresses: *handler*, *handler* + 12, and *handler* + 28. The vector at the address *handler* would be used in this code for any exception that occurs. No exceptions are expected in this example program, so the code simply jumps to the endless loop at Line 56 if an extraneous exception should happen to occur.

As mentioned above, a software interrupt is created in this example when a 1 is written into the *msip* register by the store operation in Line 47 of Figure 12*a*. In response to the corresponding pending interrupt, the processor will jump to the vector address *handler* + 12. The instruction at this address (see Line 58 of Figure 12*b*) then jumps to the label IRQ_3, which provides the software interrupt handler. After saving temporary registers that it will modify, the handler clears the pending interrupt by writing 0 into the *msip* register, in Line 70. Then, the handler writes a pattern to illuminate all ten red lights in the LEDR output port, so that a user can visually confirm that the software interrupt handler has been executed. Finally, the handler restores temporary register values and then executes mret to return to (Line 49 of) the main program.

The main program in Figure 12*a* will execute the loop between Lines 51 and 54 indefinitely. Each time the machine timer generates an interrupt, the processor will jump to the vector address *handler* + 28. The instruction at this address (see Line 60 of Figure 12*b*) then jumps to the label IRQ_7, which provides the machine timer interrupt handler. Most of the code in this handler is not shown, because it is the same as the code described previously for Figure 9, except that this handler has to save onto the stack the temporary registers that it will modify. Once this handler is completed, it restores saved temporary registers and then executes the mret instruction in Line 90 to return to the interrupted loop in the main program.