Omega CPU
Developers' Reference Manual

August Schwerdfeger

Current to revision `db7bac7`

Contents

Abstract

This manual contains documentation of the instruction-set architecture and development tools of the Omega CPU, a 32-bit RISC processor patterned after MIPS I.

It is organized as follows. Chapter 1 describes the Omega instruction-set architecture, including its register and memory model. Chapter 2 describes programming conventions for the architecture, including those for procedure calls and interrupt handling. Chapter 3 describes the Omega assembler, including several pseudo-instructions making use of the described calling convention. Chapter 4 describes the existing implementations of the architecture.

As the Omega CPU is under active development, this manual may not accurately reflect the latest changes in the architecture and/or implementations. Documentation of incomplete features will be annotated in type *like this*.

Chapter 1

Instruction-set architecture.

1.1 Overview.

Arithmetic	Integer only, unsigned or two's complement
Addressing mode	Base + offset
Endianness	Little
I/O	Port-mapped, 32 ports
Instruction length	Fixed, 32 bits
Max. operand count	4
Memory model	Flat, byte-addressable, 4 GiB max (absolute immediate jump limited to lower 256 MiB)
Registers	1 constant 28 general-purpose 1 status 2 special-purpose
Word size	32 bits

Table 1: Overview of Omega instruction-set architecture.

1.2 Registers.

The Omega architecture has 32 accessible registers, numbered 0 through 31. All are 32 bits in size and of integer type.

Register 0. Register 0 is a constant register with a fixed value of 0.
Instructions writing to register 0 are treated as no-ops.

Registers 1–28. Registers 1–28 are fully general-purpose registers (but see section 2.1.1 for the purposes assigned to them by convention).

Register 29. Register 29 is a special-purpose register used to hold a return address during interrupt handling (see section 2.2).

Instructions may write to register 29.

Register 30. Register 30 is a status register, with fields as follows:

- **Bit 0: Carry bit.** Following an add or subtract instruction, this bit will be 1 if the add or subtract operation overflowed, and 0 otherwise.
- **Bits 1–31: Unused.**

Instructions may write to register 30.

Register 31. Register 31 is the program counter.

Instructions may write to register 31, but this is strongly discouraged.

1.3 Memory model.

The Omega architecture has a flat or linear memory model. Except for an interrupt vector table (see section 2.2*, but interrupt handling is incomplete*), the whole of memory is available for general-purpose use.

The maximum amount of memory supported is the full 4 GiB allowed by a 32-bit address space. However, the absolute immediate jump instruction (documented on page 15) holds its absolute address in the 26-bit I_A field. This gives only 28 effective address bits, so this jump instruction cannot be used to jump to any address beyond the lower 256 MiB. For absolute jumps to such addresses, the absolute register jump must be used instead.

1.4 Instruction set.

See Table 2 for a tabular representation of all instructions in the Omega instruction set.

1.4.1 Formats and fields.

The Omega instruction set has a fixed instruction size of 32 bits, of which the upper 6 are the opcode and the lower 26 hold any operands. Depending on the specific *format* of the instruction (see Table 3 for a list of formats), the lower 26 bits will be interpreted as consisting of one or more of the *fields* delineated at the top of Table 2 (see Table 4 for a full list).

1.4.2 Instructions in detail.

Terminology. In this section, we describe the behavior of each instruction using snippets of pseudo-C. Some particulars of this notation:

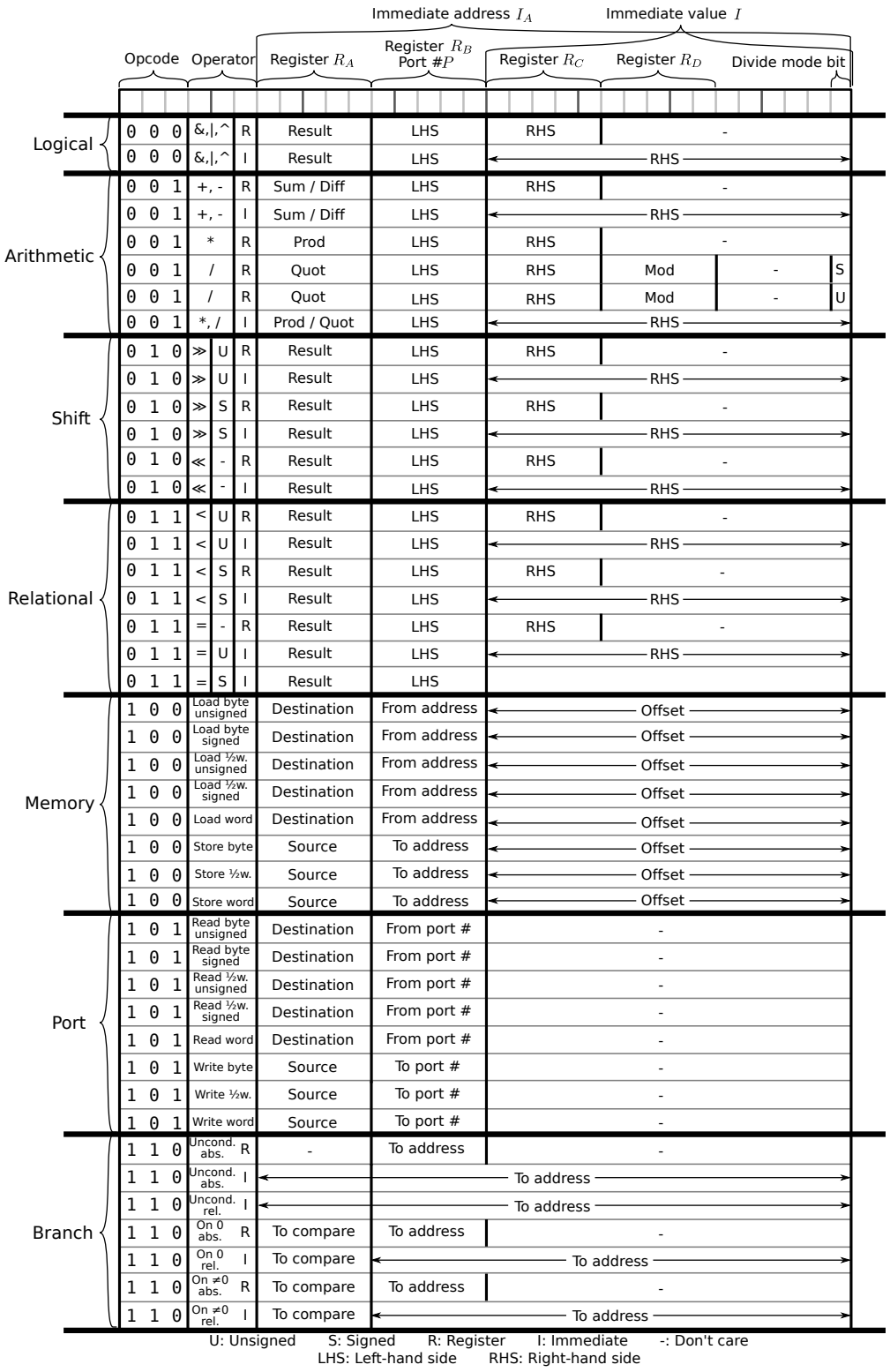

Table 2: Instruction set.

Format	Operands
IA	Immediate address only
1R	1 register
1R+IA	1 register and immediate address
1R+P	1 register and 1 port number
2R	2 registers
2R+I	2 registers and immediate value
3R	3 registers
4R+M	4 registers and mode bit

Table 3: Instruction formats.

Field	Bit span	Description
DM	0	A bit used on divide instructions to select signed or unsigned mode.
I	15–0	16-bit immediate value field.
I_A	25–0	26-bit immediate address field for unconditional jumps.
I_C	20–0	21-bit immediate address field for conditional branches.
P	20–16	5-bit port number field, overlapping R_B.
R_A	25–21	5-bit register field.
R_B	20–16	5-bit register field.
R_C	15–11	5-bit register field.
R_D	10–6	5-bit register field.

Table 4: Operand fields.

Field→ ↓Format	DM	I	I_A	I_C	P	R_A	R_B	R_C	R_D
IA			×						
1R							×		
1R+I				×		×			
1R+P					×	×			
2R						×	×		
2R+I		×				×	×		
3R						×	×	×	
4R+M	×					×	×	×	×

Table 5: Which fields are used in which instruction formats.

Operator→ ↓Opcode	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	OR **3R**	ORI **2R+I**	AND **3R**	ANDI **2R+I**	XOR **3R**	XORI **2R+I**	*Invalid* *Instr.*	*Invalid* *Instr.*
001	ADD **3R**	ADDI **2R+I**	SUB **3R**	SUBI **2R+I**	MULT **3R**	MULTI **2R+I**	DIV **4R+M**	DIVI **2R+I**
010	SRAV **3R**	SRA **2R+I**	SRLV **3R**	SRL **2R+I**	SLLV **3R**	SLL **2R+I**	*SLLV* ***3R***	*SLL* ***2R+I***
011	EQ **3R**	EQI **2R+I**	*EQ* ***3R***	EQUI **2R+I**	LT **3R**	LTI **2R+I**	LTU **3R**	LTUI **2R+I**
100	LBU **2R+I**	LB **2R+I**	LHU **2R+I**	LH **2R+I**	LW **2R+I**	SB **2R+I**	SH **2R+I**	SW **2R+I**
101	INPBU **1R+P**	INPB **1R+P**	INPHU **1R+P**	INPH **1R+P**	INP **1R+P**	OUTPB **1R+P**	OUTPH **1R+P**	OUTP **1R+P**
110	JR **1R**	JA **IA**	J **IA**	BZ **2R**	BZI **1R+I**	BNZ **2R**	BNZI **1R+I**	*NOP*
111	Invalid Instr.	Invalid Instr.	Invalid Instr.	Invalid Instr.	Invalid Instr.	Invalid Instr.	Invalid Instr.	Invalid Instr.

Table 6: Opcode table. Rows represent the opcode proper (upper 3 bits); columns, the operator (lower 3 bits). *SLLV, SLL, and EQ have duplicate opcodes, marked in this typeface, that the assembler does not use, but that the implementations recognize.*

- System memory is represented as an array `mem`; registers and other operands as variables of the type as which they are treated during the instruction's operation.
- When converting a signed integer to a type of greater bit width (signed or unsigned), sign extension is performed. When converting unsigned integers, zero-filling is used.
- Integer variables may also be addressed as arrays of bits; *e.g.* `var[7:0]` refers to the least significant byte of a word.

1.4.2.1 Logical instructions.

`OR`. Bitwise OR, register mode.

```
uint32_t RA, RB, RC;
RA = RB | RC;
```

`ORI`. Bitwise OR, immediate mode.

```
uint32_t RA, RB;
uint16_t I;
RA = RB | (uint32_t) I;
```

`AND`. Bitwise AND, register mode.

```
uint32_t RA, RB, RC;
RA = RB & RC;
```

`ANDI`. Bitwise AND, immediate mode.

```
uint32_t RA, RB;
uint16_t I;
RA = RB & (uint32_t) I;
```

`XOR`. Bitwise XOR, register mode.

```
uint32_t RA, RB, RC;
RA = RB ^ RC;
```

`XORI`. Bitwise XOR, immediate mode.

```
uint32_t RA, RB;
uint16_t I;
RA = RB ^ (uint32_t) I;
```

1.4.2.2 Arithmetic instructions.

`ADD`. Addition, register mode.

```
uint32_t R_A, R_B, R_C;
uint33_t result = R_B + R_C;
R_A = result[31:0];
Carry = result[32];
```

`ADDI`. Addition, immediate mode.

```
uint32_t R_A, R_B;
int16_t I;
uint33_t result = R_B + (uint32_t) I;
R_A = result[31:0];
Carry = result[32];
```

`SUB`. Subtraction, register mode.

```
uint32_t R_A, R_B, R_C;
uint33_t result = R_B - R_C;
R_A = result[31:0];
Carry = result[32];
```

`SUBI`. Subtraction, immediate mode.

```
uint32_t R_A, R_B;
int16_t I;
uint33_t result = R_B - (uint32_t) I;
R_A = result[31:0];
Carry = result[32];
```

`MULT`. Multiplication, register mode.

```
uint32_t R_A, R_B, R_C;
uint64_t result = R_B * R_C;
R_A = result[31:0];
```

`MULTI`. Multiplication, immediate mode.

```
uint32_t R_A, R_B;
int16_t I;
uint64_t result = R_B * (uint32_t) I;
R_A = result[31:0];
```

`DIV`. Division, register mode. It must be explicitly specified, using the DM bit, whether the operation should be signed or unsigned.

```
bool DM;
if(DM) { // Signed mode
    int32_t R_A, R_B, R_C, R_D;
} else { // Unsigned mode
    uint32_t R_A, R_B, R_C, R_D;
}
if(R_C == 0) {
    Status = DivideOverflow;
} else {
    R_A = R_B / R_C;
    R_D = R_B % R_C;
}
```

`DIVI`. Division, immediate mode. This operates in signed mode only *(but the implementation will set the mode according to the DM bit)*.

```
int32_t R_A, R_B;
int16_t I;
R_A = R_B / (int32_t) I;
```

1.4.2.3 Shift instructions.

`SRAV`. Shift right, arithmetic (sign-preserving), register mode.

```
int32_t R_A, R_B, R_C;
R_A = R_B >> R_C;
```

`SRA`. Shift right, arithmetic (sign-preserving), immediate mode.

```
int32_t R_A, R_B;
uint16_t I;
R_A = R_B >> I;
```

`SRLV`. Shift right, logical (not sign-preserving), register mode.

```
uint32_t R_A, R_B, R_C;
R_A = R_B >> R_C;
```

`SRL`. Shift right, logical (not sign-preserving), immediate mode.

```
uint32_t R_A, R_B;
uint16_t I;
R_A = R_B >> I;
```

SLLV. Shift left, register mode.

```
uint32_t R_A, R_B, R_C;
R_A = R_B << R_C;
```

SLL. Shift left, immediate mode.

```
uint32_t R_A, R_B;
uint16_t I;
R_A = R_B << I;
```

1.4.2.4 Relational instructions.

EQ. Equality test, register mode.

```
uint32_t R_A, R_B, R_C;
if(R_B == R_C) {
    R_A = 1;
} else {
    R_A = 0;
}
```

EQI. Equality test, signed, immediate mode.

```
uint32_t R_A, R_B;
int16_t I;
if(R_B == (uint32_t) I) {
    R_A = 1;
} else {
    R_A = 0;
}
```

EQUI. Equality test, unsigned, immediate mode.

```
uint32_t R_A, R_B;
uint16_t I;
if(R_B == (uint32_t) I) {
    R_A = 1;
} else {
    R_A = 0;
}
```

LT. Inequality test, signed, register mode.

```
uint32_t R_A;
int32_t R_B, R_C;
if(R_B < R_C) {
    R_A = 1;
} else {
    R_A = 0;
}
```

LTI. Inequality test, signed, immediate mode.

```
uint32_t R_A;
int32_t R_B;
int16_t I;
if(R_B < (int32_t) I) {
    R_A = 1;
} else {
    R_A = 0;
}
```

LTU. Inequality test, unsigned, register mode.

```
uint32_t R_A, R_B, R_C;
if(R_B < R_C) {
    R_A = 1;
} else {
    R_A = 0;
}
```

LTUI. Inequality test, unsigned, immediate mode.

```
uint32_t R_A, R_B;
uint16_t I;
if(R_B < (uint32_t) I) {
    R_A = 1;
} else {
    R_A = 0;
}
```

1.4.2.5 Load and store instructions.

LBU. Retrieve a byte from memory into a register, without sign extension.

```
uint32_t R_A;
uint32_t R_B;
int16_t I;
uint32_t addr = R_B + (int32_t) I;
R_A = (uint32_t) mem[addr];
```

LB. Retrieve a byte from memory into a register, with sign extension.

```
uint32_t R_A;
uint32_t R_B;
int16_t I;
uint32_t addr = R_B + (int32_t) I;
R_A = (int32_t) mem[addr];
```

LHU. Retrieve a half-word from memory into a register, without sign extension.

```
uint32_t R_A;
uint32_t R_B;
int16_t I;
uint32_t addr = R_B + (int32_t) I;
R_A[7:0] = mem[addr];
R_A[31:8] = (uint24_t) mem[addr + 1];
```

LH. Retrieve a half-word from memory into a register, with sign extension.

```
uint32_t R_A;
uint32_t R_B;
int16_t I;
uint32_t addr = R_B + (int32_t) I;
R_A[7:0] = mem[addr];
R_A[31:8] = (int24_t) mem[addr + 1];
```

LW. Retrieve a word from memory into a register.

```
uint32_t R_A;
uint32_t R_B;
int16_t I;
uint32_t addr = R_B + (int32_t) I;
R_A[7:0] = mem[addr];
R_A[15:8] = mem[addr + 1];
R_A[23:16] = mem[addr + 2];
R_A[31:24] = mem[addr + 3];
```

`SB`. Write a byte from a register into memory.

```
uint32_t R_A;
uint32_t R_B;
int16_t I;
uint32_t addr = R_B + (int32_t) I;
mem[addr] = R_A[7:0];
```

`SH`. Write a half word from a register into memory.

```
uint32_t R_A;
uint32_t R_B;
int16_t I;
uint32_t addr = R_B + (int32_t) I;
mem[addr] = R_A[7:0];
mem[addr + 1] = R_A[15:8];
```

`SW`. Write a word from a register into memory.

```
uint32_t R_A;
uint32_t R_B;
int16_t I;
uint32_t addr = R_B + (int32_t) I;
mem[addr] = R_A[7:0];
mem[addr + 1] = R_A[15:8];
mem[addr + 2] = R_A[15:8];
mem[addr + 3] = R_A[15:8];
```

1.4.2.6 I/O instructions.

`INPBU`. Read a byte from a port into a register, without sign extension. *This instruction may, at the discretion of the port controller, take an arbitrary amount of time to execute.*

```
uint32_t R_A;
uint5_t P;
R_A = (uint32_t) inp(P)[7:0];
```

`INPB`. Read a byte from a port into a register, with sign extension. *This instruction may, at the discretion of the port controller, take an arbitrary amount of time to execute.*

```
uint32_t R_A;
uint5_t P;
R_A = (int32_t) inp(P)[7:0];
```

`INPHU`. Read a half-word from a port into a register, without sign extension. *This instruction may, at the discretion of the port controller, take an arbitrary amount of time to execute.*

```
uint32_t R_A;
uint5_t P;
R_A = (uint32_t) inp(P)[15:0];
```

`INPH`. Read a half-word from a port into a register, with sign extension. *This instruction may, at the discretion of the port controller, take an arbitrary amount of time to execute.*

```
uint32_t R_A;
uint5_t P;
R_A = (int32_t) inp(P)[15:0];
```

`INP`. Read a word from a port into a register. *This instruction may, at the discretion of the port controller, take an arbitrary amount of time to execute.*

```
uint32_t R_A;
uint5_t P;
R_A = inp(P);
```

`OUTPB`. Write a byte from a register to a port.

```
uint32_t R_A;
uint5_t P;
outp(P, (uint32_t) R_A[7:0]);
```

`OUTPH`. Write a half word from a register to a port.

```
uint32_t R_A;
uint5_t P;
outp(P, (uint32_t) R_A[15:0]);
```

`OUTP`. Write a word from a register to a port.

```
uint32_t R_A;
uint5_t P;
outp(P, R_A);
```

1.4.2.7 Branch instructions.

`JR`. Unconditional jump to an absolute address stored in a register.

```
uint32_t PC;
uint32_t R_A;
PC = R_A;
```

`JA`. Unconditional jump to an absolute immediate address.

```
uint32_t PC;
int26_t I_A;
PC = (uint32_t) (I_A << 2);
```

`J`. Unconditional jump to a relative immediate address.

```
uint32_t PC;
int26_t I_A;
PC = PC + (int32_t) (I_A << 2);
```

`BZ`. Branch on a zero condition to an absolute address stored in a register.

```
uint32_t PC;
uint32_t R_A, R_B;
if(R_A == 0) {
    PC = R_B;
}
```

`BZI`. Branch on a zero condition to a relative immediate address.

```
uint32_t PC;
uint32_t R_A;
int21_t I_C;
if(R_A == 0) {
    PC = PC + (int32_t) (I_C << 2);
}
```

`BNZ`. Branch on a nonzero condition to an absolute address stored in a register.

```
uint32_t PC;
uint32_t R_A, R_B;
if(R_A != 0) {
    PC = R_B;
}
```

`BNZI`. Branch on a nonzero condition to a relative immediate address.

```
uint32_t PC;
uint32_t R_A;
int21_t I_C;
if(R_A != 0) {
    PC = PC + (int32_t) (I_C << 2);
}
```

Chapter 2

Conventions and implementation details.

2.1 Calling convention.

The calling convention implemented by the Omega assembler is patterned after the O32 convention used with MIPS.

2.1.1 Register usage.

Table 7 shows the designated purpose of registers under the calling convention.

Register	Purpose	Must be preserved through call	Programs should use directly
1	Assembler temporary	No	**No**
2,3	Return values 0,1	No	Yes
4–7	Parameters 0–3	No	Yes
8–15	General purpose (volatile)	No	Yes
16–23	General purpose (static)	**Yes**	Yes
24–26	General purpose (volatile)	No	Yes
27	Stack pointer	**Yes**	Yes
28	Frame pointer	**Yes**	Read only
29	Return address	No	**No**
30	Status register	No	Read only
31	Program counter	No	**No**

Table 7: How registers are used in the Omega calling convention; whether a called routine must leave them as found, and whether assembly-language programs should reference them directly.

2.1.2 Stack structure and protocols.

The stack in the Omega calling convention grows downward from any selected address. Both the stack and frame pointers (registers 27 and 28, respectively) are set to this address to start.

A push is performed by storing the new element to the location indicated by the stack pointer, then decrementing the stack pointer by 4. A pop is performed by incrementing the stack pointer by 4.

A new stack frame is created by pushing the value of the frame pointer and then the value of the return address register (register 29). The frame pointer is then set to point at the stack element containing its former value. When the stack frame is destroyed, these two values are popped and restored to their respective registers.

The exact mechanisms by which this convention is implemented are detailed in section 3, in the documentation of the assembler's `CALL` and `RET` pseudo-instructions.

2.2 Interrupt handling.

Although the specific details of interrupt handling are left to implementations *(and are incomplete)*, this is the common behavior to be followed by all implementations.

There is a section of memory designated as the interrupt vector table. This may begin at any memory address IV_0 chosen by the implementation, and will consist of as many words as there are available interrupts. Each interrupt will be given a unique numeric designator, greater than or equal to 1.

Before executing each instruction, the processor will check if any interrupts have been received and not serviced, and if so will service the one with the lowest number, n.

In servicing an interrupt, the processor will first load the contents of memory location $IV_0 + 4n$. If it is equal to 0, no action will be taken. But if it is non-zero, the current value of the program counter will be written to register 29 and the value at memory location $IV_0 + 4n$ will be written to the program counter.

Thereafter, no more interrupts will be serviced until after the processor executes a `JR` instruction with register 29 as the operand. This instruction is meant to be rendered in the assembler using the `RET` pseudo-instruction (see page 22).

Chapter 3

Assembler.

The Omega assembler is a Python-language cross-assembler with syntax derived from that of the MIPS assembly language.

3.1 Command-line interface.

The assembler may be invoked from the command line as follows:

```
$ python OmegaAssembler.py asm_1.s asm_2.s ...  asm_n.s >
out.bin
```

This will concatenate all the input files (`asm_1.s` through `asm_n.s`) into one string, in command-line order. This string is then assembled and the resulting machine code sent to standard output.

3.2 Syntax overview.

Each assembly-language file consists of zero or more lines.

A line may be blank (whitespace only), a comment (beginning with a hash symbol), an instruction, or a directive. Lines containing instructions and directives generally begin with a tab character.

Any line (blank or not) may be prepended with one or more alphanumeric labels. Labels must start with a letter or underscore; contain only letters, numbers, underscores, and periods; and be suffixed with a colon. Each label will be assigned the address of the most closely following data-directive or instruction (see section 3.3 for details on address assignment).

For example, in the following snippet, the labels `foo` and `bar` will both be assigned the address of the add instruction:

```
    J foo
    J bar
foo:
bar:  ADD $r1,$r2,$r3
```

Instructions and directives consist of an opcode or directive name followed by one or more operands. All supported operand types are shown in Table 8.

Operand type	Syntax	Matches regex
Immediate value	Signed decimal integer	`[-]?(0|[1-9][0-9]*)`
Label reference	Name of label	`[A-Za-z_][A-Za-z_.0-9]`
Port reference	`$p`n	N/A
Register reference	`$r`n	N/A
String literal (`.asciiz` directive only)	Python-syntax string	N/A

Table 8: Operand types supported by the Omega assembler.

3.3 Address assignment and output format.

Each instruction and data directive in the input is assigned a memory address where its corresponding machine code will be placed.

By default, the first instruction or data-directive in the input will be assigned to memory address 0, while each subsequent instruction/data-directive will be assigned to the next (word-aligned) address following its immediate predecessor. However, the `.data` and `.text` directives may specify an address explicitly.

The output of the assembler is a series of lines, each a 32-bit binary number representing a word of memory in canonical form (*i.e.*, the sequence of four bytes in each word must be reversed to produce the actual order they will take in memory). The sequence starts at address 0 and progresses as high as necessary: line n corresponds to memory address $4 \cdot (n - 1)$.

3.4 Directives and instructions.

3.4.1 Machine instructions.

The majority of Omega assembler instructions correspond exactly with machine instructions and are assembled into a single machine instruction (4 bytes in size), with some minor changes, as detailed below:

Assembler instruction	Machine instruction
`OR` R_A, R_B, R_C	`OR`
`ORI` R_A, R_B, I	`ORI`
`AND` R_A, R_B, R_C	`AND`
`ANDI` R_A, R_B, I	`ANDI`
`XOR` R_A, R_B, R_C	`XOR`

XORI R_A,R_B,I	XORI
ADD R_A,R_B,R_C	ADD
ADDI R_A,R_B,I	ADDI
SUB R_A,R_B,R_C	SUB
SUBI R_A,R_B,I	SUBI
MULT R_A,R_B,R_C	MULT
MULTI R_A,R_B,I	MULTI
DIV R_A,R_B,R_C	DIV with DM = 1
DIVU R_A,R_B,R_C	DIV with DM = 0
DIVI R_A,R_B,I	DIVI
SRAV R_A,R_B,R_C	SRAV
SRA R_A,R_B,I	SRA
SRLV R_A,R_B,R_C	SRLV
SRL R_A,R_B,I	SRL
SLLV R_A,R_B,R_C	SLLV
SLL R_A,R_B,I	SLL
EQ R_A,R_B,R_C	EQ
EQI R_A,R_B,I	EQI
EQUI R_A,R_B,I	EQUI
LT R_A,R_B,R_C	LT
LTI R_A,R_B,I	LTI
LTU R_A,R_B,R_C	LTU
LTUI R_A,R_B,I	LTUI
LBU R_A,R_B[,I]	LBU (I = 0 if omitted)
LB R_A,R_B[,I]	LB (I = 0 if omitted)
LHU R_A,R_B[,I]	LHU (I = 0 if omitted)
LH R_A,R_B[,I]	LH (I = 0 if omitted)
LW R_A,R_B[,I]	LW (I = 0 if omitted)
SB R_A,R_B[,I]	SB (I = 0 if omitted)
SH R_A,R_B[,I]	SH (I = 0 if omitted)
SW R_A,R_B[,I]	SW (I = 0 if omitted)
INPBU R_A,P	INPBU
INPB R_A,P	INPB
INPHU R_A,P	INPHU
INPH R_A,P	INPH
INP R_A,P	INP
OUTPB R_A,P	OUTPB
OUTPH R_A,P	OUTPH
OUTP R_A,P	OUTP
JR R_A	JR
JA *label*	JA with I_A = absolute addr. of *label*
J *label*	J with I_A = relative addr. of *label*

BZ R_A,R_B	BZ
BZI R_A,*label*	BZI with I_C = relative addr. of *label*
BNZ R_A,R_B	BNZ
BNZI R_A,*label*	BNZI with I_C = relative addr. of *label*

3.4.2 Pseudo-instructions.

There are three Omega assembler instructions that do not correspond to any machine instruction, and are assembled into more than one machine instruction.

Programs using any of these pseudo-instructions are expected to follow the conventions for register usage detailed in Table 7.

LA.

Form. LA R, *label*

Assembled length. 28 bytes.

Function. Load the absolute address of *label* into the register R.

Machine code. If the address of *label* is held in a 32-bit integer addr, the pseudo-instruction LA R, *label* is converted to the following machine code:

```
ADDI R,$r0,addr[31:24]
SLL R,R,8
ADDI R,R,addr[23:16]
SLL R,R,8
ADDI R,R,addr[15:8]
SLL R,R,8
ADDI R,R,addr[7:0]
```

CALL.

Form. CALL *label*

Assembled length. 44 bytes.

Function. Call a subroutine, according to the calling convention detailed above.

Machine code. If the address of *label* is held in a 26-bit integer `addr`, the pseudo-instruction `CALL` *label* is converted to the following machine code:

```
SW $r28,$r27
ADDI $r28,$r27,0
SUBI $r27,$r27,4
SW $r29,$r27
SUBI $r27,$r27,4
ADDI $r29,$r31,4
J addr
SUBI $r27,$r28,4
LW $r29,$r27
ADDI $r27,$r27,4
LW $r28,$r27
```

`RET`**.**

Form. `RET` (no operands)

Assembled length. 4 bytes.

Function. Return from a function called according to the calling convention detailed above.

Machine code. A `RET` pseudo-instruction translates to a single machine instruction:

```
JR $r29
```

3.4.3 Directives.

`.asciiz`**.**

Form. `.asciiz` *string*

Assembled length. String length + 1.

Function. Put a null-terminated ASCII string at the designated location in memory. The parameter is parsed within the assembler as a Python string literal and supports all escape sequences.

`.byte`**.**

Form. `.byte` *I*{`,`*I*}

Assembled length. Number of operands.

Function. Put one or more data bytes at the designated location in memory, in the exact order given. The numbers must be in decimal form and in the range 0 to 255.

`.data`**.**

Form. `.data [`*I*`]`

Assembled length. N/A.

Function. Indicates the start of a data segment. If the operand *I* is provided *(but this does not work)*, the segment will start at the given address *(or the next higher multiple of 4)*.

`.text`**.**

Form. `.text [`*I*`]`

Assembled length. N/A.

Function. Indicates the start of a code segment. If the operand *I* is provided, the segment will start at the given address *(or the next higher multiple of 4)*.

Chapter 4

Implementations.

4.1 Papilio Duo FPGA board.

4.2 Emulation in GHDL.

4.3 Emulation in software.