Overpowered Assembly Language
Technical Reference Manual
Revision 1.0

Andy Meyer
Wesley Dahar

Contents

1 Architecture **2**
- 1.1 Overview . . . 2
- 1.2 Hardware Overview and Cycle Time . . . 3
- 1.3 Memory . . . 3
- 1.4 Registers . . . 4

2 Instruction Set **5**
- 2.1 Overview . . . 5
- 2.2 Instructions . . . 6
  - 2.2.1 ADD - Add . . . 6
  - 2.2.2 AND - Bitwise AND . . . 7
  - 2.2.3 CMP - Compare . . . 8
  - 2.2.4 HCF - Halt and Catch Fire . . . 9
  - 2.2.5 INV - Bitwise NOT . . . 10
  - 2.2.6 JAP - Jump and Push . . . 11
  - 2.2.7 J<Suffix> - Conditional Jumps . . . 12
  - 2.2.8 LDA - Load Address . . . 14
  - 2.2.9 LDAB - Load Address Byte . . . 15
  - 2.2.10 OR - Bitwise OR . . . 16
  - 2.2.11 S<Suffix> - Shift Operations . . . 17
  - 2.2.12 SGX - Sign Extend Byte . . . 18
  - 2.2.13 STA - Store Accumulator . . . 19
  - 2.2.14 STAB - Store Accumulator Byte . . . 20
  - 2.2.15 SUB - Subtract . . . 21
  - 2.2.16 SWP - Swap with Accumulator . . . 22
  - 2.2.17 XOR - Bitwise XOR . . . 23

3 Standardized Macros **24**
- 3.1 Overview . . . 24
- 3.2 Macros . . . 25
  - 3.2.1 DEC - Decrement . . . 25
  - 3.2.2 INC - Increment . . . 26
  - 3.2.3 JCC - Jump if C-Flag Clear . . . 27
  - 3.2.4 JCS - Jump if C-Flag Set . . . 28
  - 3.2.5 JZC - Jump if Z-Flag Clear . . . 29
  - 3.2.6 JZS - Jump if Z-Flag Set . . . 30
  - 3.2.7 LDI - Load Immediate . . . 31
  - 3.2.8 LDIB - Load Immediate Byte . . . 32
  - 3.2.9 RET - Return . . . 33

4 Diagrams and Programs **34**
- 4.1 Example Control Signal Enumeration . . . 38
- 4.2 Sample Program . . . 39

1 Architecture

1.1 Overview

The Overpowered Assembly Language (OPAL) is an accumulator architecture. All logical and memory read operations will write to the Accumulator, and all memory write operations will read from the Accumulator. Additionally, there are three Argument Registers which can be used as instruction operands, but they can only be written to by swapping values with the Accumulator. The third Argument Register, the Stack Pointer, is named according to design convention and also used to preserve the Program Counter for subroutine calls. The Program Counter is not an addressable register and can only be written to to or read from by select control instructions. The Flags register is also not addressable and is only modified when the Accumulator is written to by either logical or memory read operations. It is used by the control instructions for the purpose of conditional execution.

OPAL instructions are 16-bit words, aligned on a 2-byte boundary. All instructions that support operands can make use of the 4 addressable registers and up to a 16-bit immediate. In general, instructions that specify two operands (either a register and an immediate or two registers) will overwrite the value in the Accumulator, while instructions that only specify one operand will use the Accumulator in place of the first. A few notable exceptions are Swap, Sign Extend, and Load Immediate, which only support one or fewer operands.

Many operations of seemingly different categorization can be performed by similar instructions. For this reason, a handful of macros are specified and expected to be included as standardized OPAL instruction syntax in all complying development environments and assemblers.

1.2 Hardware Overview and Cycle Time

The OPAL architecture is designed to have a small but powerful footprint in modern silicon processes. Since OPAL aims to be fast and low power, a minimal amount of hardware is needed to implement the device. To fully implement the 1.0 specification, two adders, two shift units, four 16-bit registers, one four-bit register, and one xor,and,or,invert module is needed. Instruction fetch and decode logic is also needed, as well as a basic memory-access unit. If OPAL units are used in a highly-parallel system, these units may be abstracted from each core, further reducing the layout area required to achieve the desired performance.

Overpowered Assembly Language is not pipelined, in keeping with the goal of small silicon footprint. Due to this restriction, cycle times are very high compared to many other modern devices. The worst case for a single device is 6 nanoseconds along the critical path during a load instruction, which requires involvement of all OPAL components. Instruction Fetch requires two nanoseconds, Instruction Decode and Register Writes each take one nanosecond, and Memory Reads take two nanoseconds.

Due to the simple nature of instruction decoding and the ability to swap register addresses within the register file, pipelining OPAL would require only a small overhead in silicon area, but provide significant reduction in cycle time. However, the pipelining of OPAL cores would introduce unpredictability for the programmer, and require advanced branch prediction in order to reduce lost cycles due to pipeline flushes and refills.

1.3 Memory

Overpowered Assembly Language supports a maximum of 64kB of byte-addressable memory for a total of 2^{16} addressable bytes.

Memory read and write operations may only read from or write to the accumulator, in order to reduce layout complexity.

1.4 Registers

OPAL defines 6 registers, 4 of which are addressable for operand use:

Register	Width	ID
A	16	00
X	16	01
Y	16	10
SP	16	11
PC	16	Not Addressable
F	4	Not Addressable

Accumulator (A)

The Accumulator is written to by all logical and memory read operations, and is read from by all memory write operations.

Argument Registers (X and Y)

The Argument Registers are used to preserve the Accumulator without having to go through memory and to conveniently order the operands for any given instruction. These register can only be written to by swapping values with the Accumulator.

Stack Pointer (SP)

The Stack Pointer is a special Argument Register. It is intended to be responsible for managing program scope and stack memory. No stack is implicitly defined or created; this task is left to the programmer. What separates the Stack Pointer from the Argument Registers X and Y is its role in the execution of the Jump and Push (JAP) instruction. Specifically, it is used as the address at which the Program Counter is stored before the jump is performed. Since it is an Argument Register, it can only be written to by swapping with the Accumulator.

Program Counter (PC)

The Program Counter is used by the Instruction Fetch Unit to load new instructions from memory for execution. Before each instruction is executed, the PC is auto-incremented by 2. Instructions which require a 16-bit immediate will increment the PC by an additional 2 after execution, to prevent that same immediate from being executed as an instruction. The Program Counter can only be modified directly by control instructions.

Flags (F)

The Flags register consists of 4 CPU state flags. These flags are used by control instructions to determine whether or not to perform a jump. The Flags are modified by logical and memory load operations, or more simply, modified whenever the Accumulator has been written to. The Flags cannot be read or written to directly.

N	Result was negative
Z	Result was zero
C	Result carried
V	Result overflowed

2 Instruction Set

2.1 Overview

Every Overpowered Assembly Language instruction is listed in this section. A comprehensive list of instructions is provided, defining their function, syntax, and binary encodings.

2.2 Instructions

2.2.1 ADD - Add

This instruction adds two values together and writes the result to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`add {<Ra>,}{#<imm8>|#<imm16>|<Rb>}`

`{<Ra>,}`	An optional register addend [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register addend [A-SP].
`<imm8>`	An 8-bit immediate addend [0-255].
`<imm16>`	A 16-bit immediate addend [0-65535].

Encoding

`add {<Ra>,}#<imm8>`

A ⟵ Ra + ZeroExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Ra		0	imm8							

`add {<Ra>,}#<imm16>`

A ⟵ Ra + M[PC, 2]
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Ra		1	0	x	x	x	x	x	x	x

`add {<Ra>,}<Rb>`

A ⟵ Ra + Rb

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Ra		1	1	Rb		x	x	x	x	x

2.2.2 AND - Bitwise AND

This instruction performs a bitwise AND with the two operands and writes the result to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`and {<Ra>,}{#<imm8>|#<imm16>|<Rb>}`

`{<Ra>,}`	An optional register operand [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register operand [A-SP].
`<imm8>`	An 8-bit immediate operand [0-255].
`<imm16>`	A 16-bit immediate operand [0-65535].

Encoding

`and {<Ra>,}#<imm8>`

A ⟵ Ra & ZeroExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Ra		0	imm8							

`and {<Ra>,}#<imm16>`

A ⟵ Ra & M[PC, 2]
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Ra		1	0	x	x	x	x	x	x	x

`and {<Ra>,}<Rb>`

A ⟵ Ra & Rb

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Ra		1	1	Rb		x	x	x	x	x

2.2.3 CMP - Compare

This instruction subtracts two values and does not store the result. It updates the Flags based on the result.

Assembler Syntax

`cmp {<Ra>,}{#<imm8>|#<imm16>|<Rb>}`

`{<Ra>,}`	An optional register minuend [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register subtrahend [A-SP].
`<imm8>`	An 8-bit immediate subtrahend [0-255].
`<imm16>`	A 16-bit immediate subtrahend [0-65535].

Encoding

`cmp {<Ra>,}#<imm8>`

Ra - ZeroExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	Ra		0	imm8							

`cmp {<Ra>,}#<imm16>`

Ra - M[PC, 2]
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	Ra		1	0	x	x	x	x	x	x	x

`cmp {<Ra>,}<Rb>`

Ra - Rb

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	Ra		1	1	Rb		x	x	x	x	x

2.2.4 HCF - Halt and Catch Fire

This instruction prevents the program counter from incrementing, causing this instruction to be repeatedly executed. It does not update the Flags.

Assembler Syntax

`hcf`

Encoding

`hcf`

PC ⟵ PC - 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	x	x	x	x	x	x	x	x	x	x	x

2.2.5 INV - Bitwise NOT

This instruction performs a bitwise NOT on a register or immediate and writes the result to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`inv {#<imm8>|#<imm16>|{<Ra>}}`

`<imm8>`	An 8-bit immediate operand [0-256].
`<imm16>`	A 16-bit immediate operand [0-65535].
`{<Ra>}`	An optional register operand [A-SP]. If `<Ra>` is omitted, the Accumulator is used.

Encoding

`inv #<imm8>`

A ⟵ ~ZeroExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	x	x	0	imm8							

`inv #<imm16>`

A ⟵ ~M[PC, 2]
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	x	x	1	0	x	x	x	x	x	x	x

`inv {<Ra>}`

A ⟵ ~Ra

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	x	x	1	1	Ra		x	x	x	x	x

2.2.6 JAP - Jump and Push

This instruction preserves the PC by writing it to the address specified by the Stack Pointer and writes to the Program Counter. It does not update the flags.

Assembler Syntax

`jap {#<imm16>|{Ra}}`

`<imm16>`	A 16-bit unsigned immediate address [0-65535].
`{<Ra>}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used.

Encoding

`jap #<imm16>` `where: imm8 = imm16 - PC`
`and: imm8 ∈ [-128, 127])`

M[SP, 2] ⟵ PC
PC ⟵ PC + SignExtend(imm16 - PC, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	x	x	0	imm8							

`jap #<imm16>` `where: imm16 - PC ∉ [-128, 127]`

M[SP, 2] ⟵ PC + 2
PC ⟵ M[PC, 2]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	x	x	1	0	x	x	x	x	x	x	x

`jap {<Ra>}`

M[SP, 2] ⟵ PC
PC ⟵ Ra

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	x	x	1	1	Ra		x	x	x	x	x

2.2.7 J<Suffix> - Conditional Jumps

These instructions evaluate the Flags and will write to the Program Counter only if the condition determined by their suffixes is met. It does not update the flags.

Assembler Syntax

`j<suffix> {#<imm16>|{Ra}}`

`<imm16>`	A 16-bit immediate address [0-65535].
`{<Ra>}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used.

Encoding

`j<suffix> #<imm16>` `where: imm8 = imm16 - PC`
`and: imm8 ∈ [-128, 127])`

M[SP, 2] ⟵ PC
PC ⟵ PC + SignExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Jump Code				x	x	0	imm8							

`j<suffix> #<imm16>` `where: imm16 - PC ∉ [-128, 127]`

M[SP, 2] ⟵ PC + 2
PC ⟵ M[PC, 2]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Jump Code				x	x	1	0	x	x	x	x	x	x	x

`j<suffix> {<Ra>}`

M[SP, 2] ⟵ PC
PC ⟵ Ra

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Jump Code				x	x	1	1	Ra		x	x	x	x	x

Suffix Codes

Each suffix is identified by a 4-bit jump code within the opcode.† The logical identities of the jump conditions and the required Flags state are given.††

Suffix	Jump Code	Condition	Flags
nc	0000	Negative Flag Clear	N = 0
ns	0001	Negative Flag Set	N = 1
ne	0010	Not Equal	Z = 0
eq	0011	Equal	Z = 1
vc	0110	Overflow Flag Clear	V = 0
vs	0111	Overflow Flag Set	V = 1
lo	0100	Lower Than (Unsigned)	C = 0
hs	0101	Higher Than or Same (Unsigned)	C = 1
hi	1010	Higher Than (Unsigned)	C = 1 and Z = 0
ls	1011	Lower Than or Same (Unsigned)	C = 0 or Z = 1
lt	1001	Less Than (Signed)	N != V
ge	1000	Greater Than or Equal (Signed)	N = V
gt	1100	Greater Than (Signed)	N = V and Z = 0
le	1101	Less Than or Equal (Signed)	N != V or Z = 1
mp	1110	Unconditional	

† The instruction JAP uses the unlisted jump code 1111.
†† Some jump codes have multiple logical identities. See the Macros in Section 3.2 for these suffixes.

2.2.8 LDA - Load Address

This instruction loads a word from memory at the specified address and writes it to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`lda [{<Ra>,}{<imm8>|<imm16>|<Rb>}]`

`{<Ra>}`	An optional register address[A-SP]. If `<Ra>` is omitted, the Accumulator is used
`<Rb>`	A register operand [A-SP].
`<imm8>`	An 8-bit immediate offset [0-256].
`<imm16>`	A 16-bit immediate offset [0-65535].

Encoding

`lda [{<Ra>,}#<imm8>]`

Ra ⟵ M[Ra + ZeroExtend(imm8, 1), 2]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Ra		0	imm8							

`lda [{<Ra>,}#<imm16>]`

Ra ⟵ M[Ra + M[PC, 2], 2]
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Ra		1	0	x	x	x	x	x	x	x

`lda [{<Ra>,}<Rb>]`

Ra ⟵ M[Ra + Rb, 2]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Ra		1	1	Rb		x	x	x	x	x

2.2.9 LDAB - Load Address Byte

This instruction loads a byte from memory at the specified address, and writes it to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`ldab [{<Ra>,}{#<imm8>|#<imm16>|<Rb>}]`

`{<Ra>,}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used
`<Rb>`	A register operand [A-SP].
`<imm8>`	An 8-bit immediate offset [0-256].
`<imm16>`	A 16-bit immediate offset [0-65535].

Encoding

`ldab [{<Ra>,}#<imm8>]`

Ra ⟵ ZeroExtend(M[Ra + ZeroExtend(imm8, 1), 1], 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	Ra		0	imm8							

`ldab [{Ra,}#<imm16>]`

Ra ⟵ ZeroExtend(M[Ra + M[PC, 2], 1], 1)
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	Ra		1	0	x	x	x	x	x	x	x

`ldab [{<Ra>,}<Rb>]`

Ra ⟵ ZeroExtend(M[Ra + Rb, 1], 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	Ra		1	1	Rb		x	x	x	x	x

2.2.10 OR - Bitwise OR

This instruction performs a bitwise OR with the two operands and writes the result to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`or {<Ra>,}{#<imm8>|#<imm16>|<Rb>}`

`{<Ra>,}`	An optional register operand [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register operand [A-SP].
`<imm8>`	An 8-bit immediate operand [0-255].
`<imm16>`	A 16-bit immediate operand [0-65535].

Encoding

`or {<Ra>,}#<imm8>`

A ⟵ Ra | ZeroExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Ra		0	imm8							

`or {<Ra>,}#<imm16>`

A ⟵ Ra | M[PC, 2]
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Ra		1	0	x	x	x	x	x	x	x

`or {<Ra>,}<Rb>`

A ⟵ Ra | Rb

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Ra		1	1	Rb		x	x	x	x	x

2.2.11 S<Suffix> - Shift Operations

These instructions cause a register to be shifted by a specified amount in a manner defined by the suffix and writes the result to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`s<suffix> {<Ra>,}{<imm16>|<Rb>}`

`{<Ra>,}`	An optional register operand [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register shift amount [A-SP].
`<imm16>`	A 16-bit immediate shift amount [0-65535].

Encoding

`s<suffix> {<Ra>,}#<imm16>`

A ⟵ Ra <shift op.> imm16[3:0]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	Ra		0	p	x	x	q	imm16[3:0]			

`s<suffix> {<Ra>,}<Rb>`

A ⟵ Ra <shift op.> Rb

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	Ra		1	p	Rb		q	x	x	x	x

Suffix Codes

Each suffix is identified by a 2-bit shift code pq. The logical identity of the shift operation is given. Fields p and q determine which type of shift should be performed.

Suffix	pq	Operation
sll	00	Shift Left Logical
srl	01	Shift Right Logical
sra	10	Shift Right Arithmetic
srr	11	Shift Right Rotate

2.2.12 SGX - Sign Extend Byte

This instruction replaces bits 8 through 15 with the value of bit 7 of the operand and stores the result in the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`sgx {#<imm8>|#<imm16>|{<Ra>}}`

`<imm8>`	An immediate operand [0-255].
`<imm16>`	An immediate operand [0-65535].
`{<Ra>}`	An optional register operand [A-SP]. If `<Ra>` is omitted, the Accumulator is used.

Encoding

`sgx #<imm8>`

A ⟵ SignExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	Ra		0	imm8							

`sgx #<imm16>`

A ⟵ SignExtend(M[PC, 2], 1)
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	Ra		1	0	x	x	x	x	x	x	x

`sgx {<Ra>}`

A ⟵ SignExtend(Ra, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	Ra		1	1	Rb		x	x	x	x	x

2.2.13 STA - Store Accumulator

This instruction writes the Accumulator to the word at the specified address. It does not update the Flags.

Assembler Syntax

`sta [{<Ra>,}{#<imm8>|#<imm16>},{<Rb>}]`

`{<Ra>,}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register addend [A-SP].
`<imm8>`	An immediate offset [0-255].
`<imm16>`	An immediate offset [0-65535].

Encoding

`sta [{<Ra>,}#<imm8>]`

M[Ra + ZeroExtend(imm8, 1), 2] ⟵ A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	Ra		0	imm8							

`sta [{<Ra>,}#<imm16>]`

M[Ra + M[PC, 2], 2] ⟵ A
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	x	x	1	0	x	x	x	x	x	x	x

`sta [{<Ra>,}<Rb>]`

M[Ra + Rb, 2] ⟵ A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	Ra		1	1	Rb		x	x	x	x	x

2.2.14 STAB - Store Accumulator Byte

This instruction writes the low byte of the Accumulator to the byte at the specified address. It does not update the Flags.

Assembler Syntax

`stab [{<Ra>,}{#<imm8>|#<imm16>|<Rb>}]`

`{<Ra>,}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register offset [A-SP].
`<imm8>`	An immediate offset [0-255].
`<imm16>`	An immediate offset [0-65535].

Encoding

`stab [{<Ra>,}#<imm8>]`

M[Ra + ZeroExtend(imm8, 1), 1] ⟵ A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	Ra		0	imm8							

`stab [{<Ra>,}#<imm16>]`

M[Ra + M[PC, 2], 1] ⟵ A
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	Ra		1	0	x	x	x	x	x	x	x

`stab [{<Ra>,}<Rb>]`

M[Ra + Rb, 1] ⟵ A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	Ra		1	1	Rb		x	x	x	x	x

2.2.15 SUB - Subtract

This instruction subtracts two values and writes the result to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`sub {<Ra>,}{#<imm8>|#<imm16>|<Rb>}`

`{<Ra>,}`	An optional register minuend [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register subtrahend [A-SP].
`<imm8>`	An immediate subtrahend [0-255].
`<imm16>`	An immediate subtrahend [0-65535].

Encoding

`sub {<Ra>,}#<imm8>`

A ⟵ Ra - ZeroExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	Ra		0	imm8							

`sub {<Ra>,}#<imm16>`

A ⟵ Ra - M[PC, 2]
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	Ra		0	1	x	x	x	x	x	x	x

`sub {<Ra>,}<Rb>`

A ⟵ Ra - Rb

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	Ra		1	1	Rb		x	x	x	x	x

2.2.16 SWP - Swap with Accumulator

This instruction swaps the value of any register with that of the Accumulator. It updates the Flags based on the result in the Accumulator.

Assembler Syntax

`swp {<Ra>}`

`<Ra>` An optional register [A-SP] to swap values with the Accumulator. If `<Ra>` is omitted, the Accumulator is used.

Encoding

`swp {<Ra>}`

A ⟵ Ra;Ra ⟵ A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Ra		x	x	x	x	x	x	x	x	x

2.2.17 XOR - Bitwise XOR

This instruction performs a bitwise XOR with the two operands and writes the result to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`xor {<Ra>,}{#<imm8>|#<imm16>|<Rb>}`

`{<Ra>,}`	An optional register operand [A-SP]. If `<Ra>` is omitted, the Accumulator is used.
`<Rb>`	A register operand [A-SP].
`<imm8>`	An 8-bit immediate operand [0-255].
`<imm16>`	A 16-bit immediate operand [0-65535].

Encoding

`xor {<Ra>,}#<imm8>`

A ⟵ Ra ⊕ ZeroExtend(imm8, 1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Ra		0	imm8							

`xor {<Ra>,}#<imm16>`

A ⟵ Ra ⊕ M[PC, 2]
PC ⟵ PC + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Ra		1	0	x	x	x	x	x	x	x

`xor {<Ra>,}<Rb>`

A ⟵ Ra ⊕ Rb

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Ra		1	1	Rb		x	x	x	x	x

3 Standardized Macros

3.1 Overview

In the course of normal system development, there comes a need for certain instructions which are not included in the ISA in use, but that could be encoded as a sequence or alias. In order to improve the clarity and readability of Overpowered Assembly Language programs, a set of standard macros and instruction aliases has been included in the specification.

3.2 Macros

3.2.1 DEC - Decrement

This instruction decrements the accumulator by one.

Assembler Syntax

```
dec
```

Equivalent Syntax

```
sub #1
```

A ⟵ A - 1

3.2.2 INC - Increment

This instruction increments the accumulator by one.

Assembler Syntax

```
inc
```

Equivalent Syntax

```
add #1
```

A ⟵ A + 1

3.2.3 JCC - Jump if C-Flag Clear

This instruction jumps to the specified address if the result of the previous operation did not carry in or out.

Assembler Syntax

`jcc {#<imm16>|{Ra}}`

`<imm16>`	A 16-bit immediate address [0-65535].
`{<Ra>}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used.

Equivalent Syntax

`jlo #<imm16>` `where: imm16 - PC` $\in$ `[-128, 127]`

PC $\longleftarrow$ PC + SignExtend(imm16 - PC, 1)

`jlo #<imm16>` `where: imm16 - PC` $\notin$ `[-128, 127]`

PC $\longleftarrow$ M[PC, 2]

`jlo {<Ra>}`

PC $\longleftarrow$ Ra

3.2.4 JCS - Jump if C-Flag Set

This instruction jumps to the specified address if the result of the previous operation did carry in or out.

Assembler Syntax

`jcs {#<imm16>|{Ra}}`

`<imm16>`	A 16-bit immediate address [0-65535].
`{<Ra>}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used.

Equivalent Syntax

`jhs #<imm16>` `where: imm16 - PC` $\in$ `[-128, 127]`

PC $\longleftarrow$ PC + SignExtend(imm16 - PC, 1)

`jhs #<imm16>` `where: imm16 - PC` $\notin$ `[-128, 127]`

PC $\longleftarrow$ M[PC, 2]

`jhs {<Ra>}`

PC $\longleftarrow$ Ra

3.2.5 JZC - Jump if Z-Flag Clear

This instruction jumps to the specified address if the result of the previous operation was not zero.

Assembler Syntax

`jzc {#<imm16>|{Ra}}`

`<imm16>`	A 16-bit immediate address [0-65535].
`{<Ra>}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used.

Equivalent Syntax

`jne #<imm16>` `where: imm16 - PC` $\in$ `[-128, 127]`

PC $\longleftarrow$ PC + SignExtend(imm16 - PC, 1)

`jne #<imm16>` `where: imm16 - PC` $\notin$ `[-128, 127]`

PC $\longleftarrow$ M[PC, 2]

`jne {<Ra>}`

PC $\longleftarrow$ Ra

3.2.6 JZS - Jump if Z-Flag Set

This instruction jumps to the specified address if the result of the previous operation was zero.

Assembler Syntax

`jzs {#<imm16>|{Ra}}`

`<imm16>`	A 16-bit immediate address [0-65535].
`{<Ra>}`	An optional register address [A-SP]. If `<Ra>` is omitted, the Accumulator is used.

Equivalent Syntax

`jeq #<imm16>` `where: imm16 - PC` $\in$ `[-128, 127]`

PC $\longleftarrow$ PC + SignExtend(imm16 - PC, 1)

`jeq #<imm16>` `where: imm16 - PC` $\notin$ `[-128, 127]`

PC $\longleftarrow$ M[PC, 2]

`jeq {<Ra>}`

PC $\longleftarrow$ Ra

3.2.7 LDI - Load Immediate

This macro writes a 16-bit immediate to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`ldi #<imm16>`

`<imm16>` A 16-bit immediate operand [0-65535].

Equivalent Syntax

`inv #(~<imm16>)`

A ⟵ ~M[PC, 2]

3.2.8 LDIB - Load Immediate Byte

This macro writes an 8-bit immediate to the Accumulator. It updates the Flags based on the result.

Assembler Syntax

`ldi #<imm8>`

`<imm8>` An 8-bit immediate operand [0-256].

Equivalent Syntax

`inv #(~<imm8>)`

A ⟵ ~ZeroExtend(imm8, 1)

3.2.9 RET - Return

This instruction returns to the previous calling function by jumping to a register value loaded from the stack. Use of `ret` assumes that `jap` was used to enter the function, which would place the PC for the next instruction from the previous function.

Assembler Syntax

```
ret
```

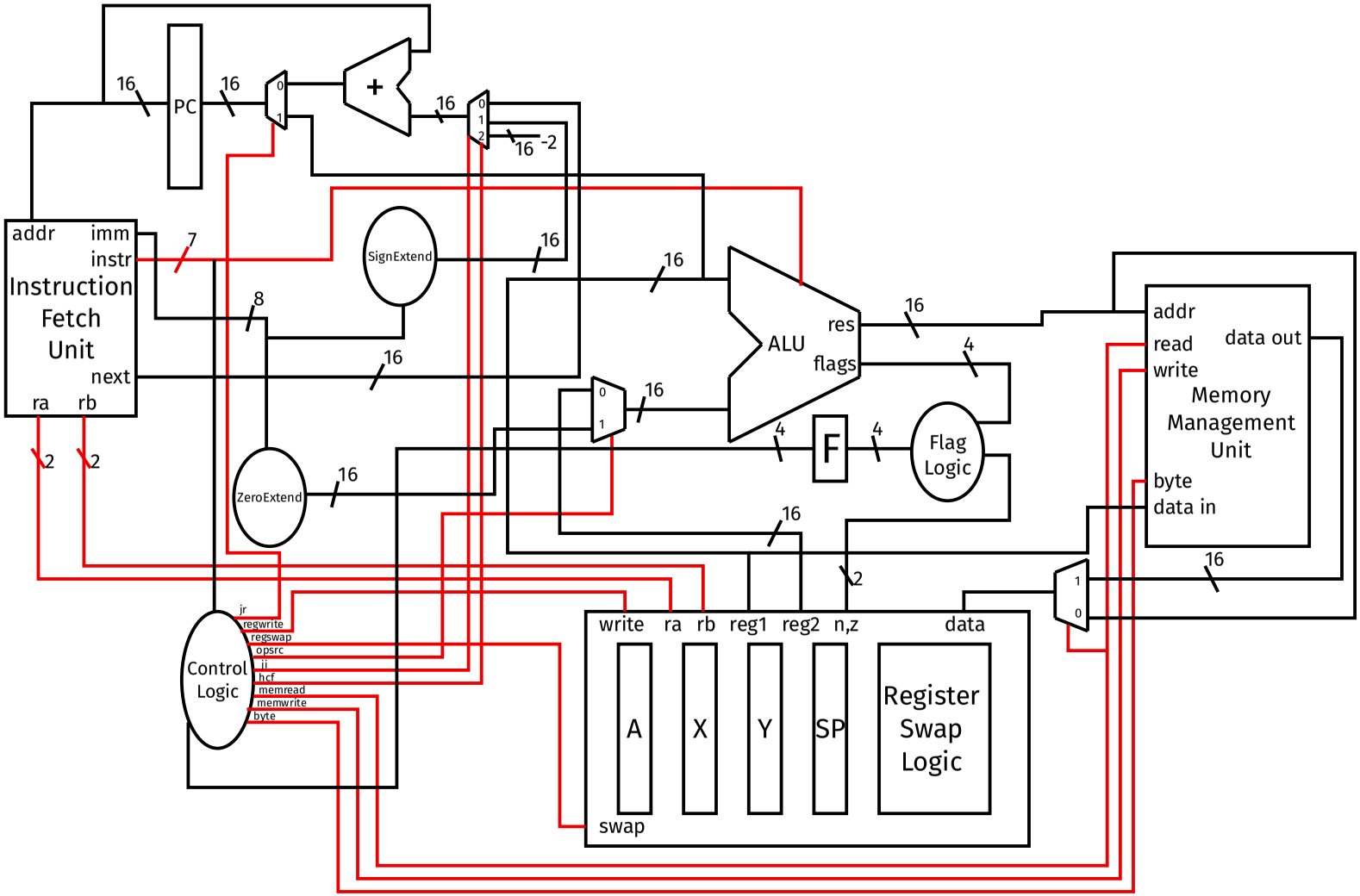
Equivalent Syntax

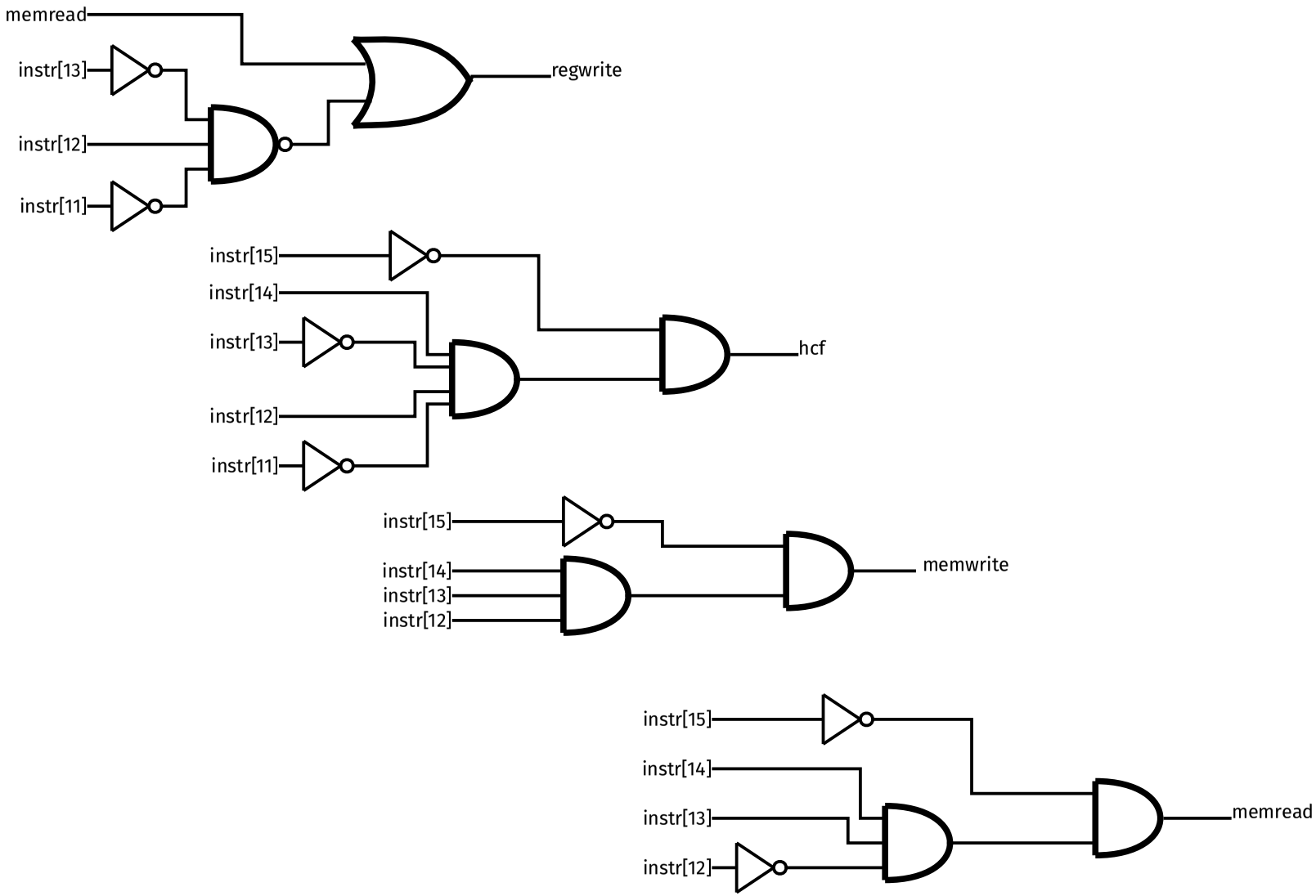
```
lda [sp, 2]
jmp
```

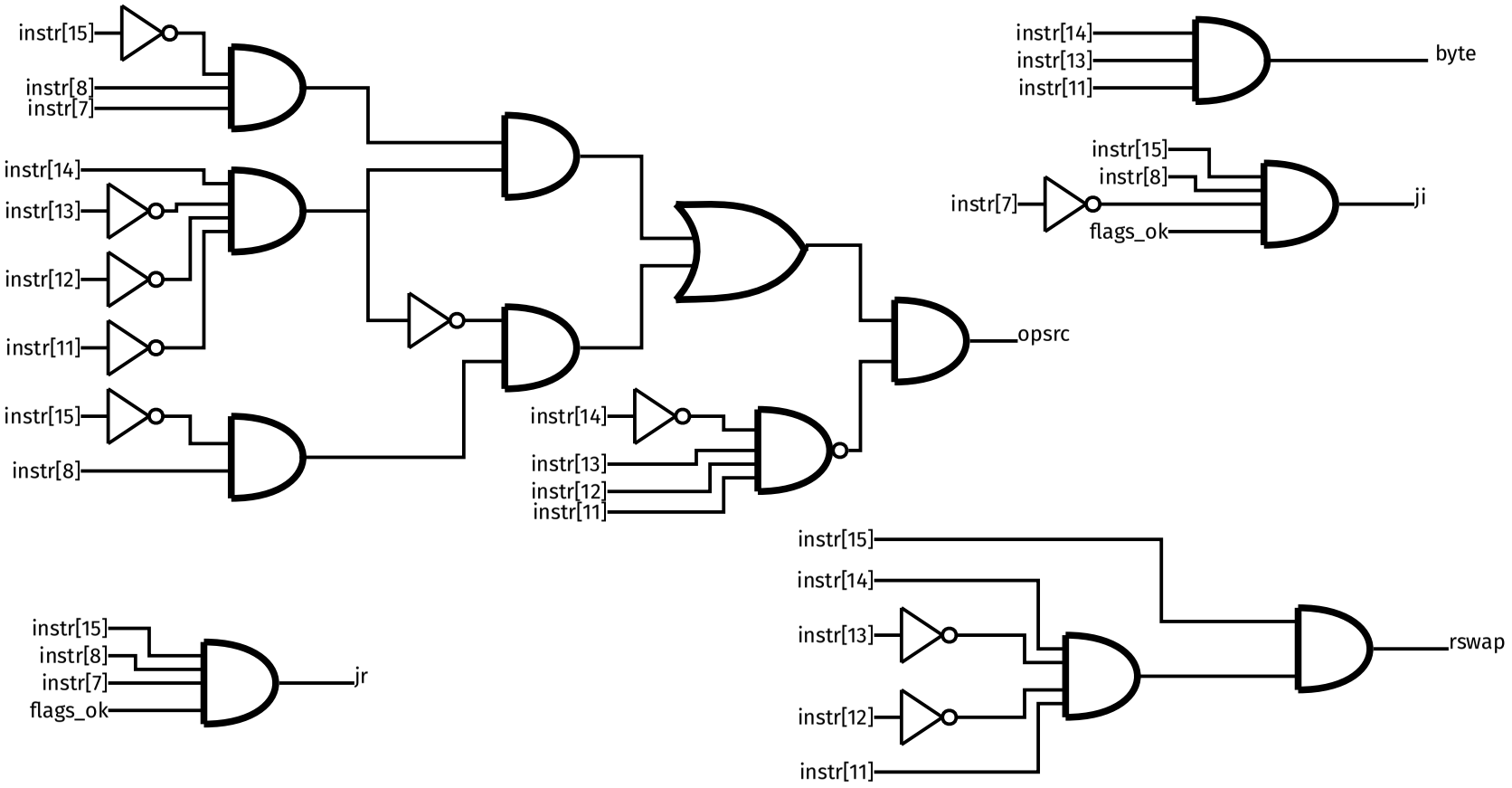
PC ⟵ M[SP + 2, 2]

4 Diagrams and Programs

The datapath and control logic signals for a subset of OPAL-1.0 has been developed to facilitate understanding of physical implementation of the design.


PC
16
+
-2
addr
imm
instr
Instruction
Fetch
Unit
next
ra
rb
7
8
SignExtend
ZeroExtend
ALU
res
flags
F
Flag
Logic
4
2
addr
read
write
data out
Memory
Management
Unit
byte
data in
Control
Logic
jr
regwrite
regswap
opsrc
ii
hcf
memread
memwrite
byte
write
ra
rb
reg1
reg2
n,z
data
A
X
Y
SP
Register
Swap
Logic
swap


memread
instr[13]
instr[12]
instr[11]
regwrite
instr[15]
instr[14]
instr[13]
instr[12]
instr[11]
hcf
instr[15]
instr[14]
instr[13]
instr[12]
memwrite
instr[15]
instr[14]
instr[13]
instr[12]
memread



flags_ok is the result of flag comparisons dependent on jump suffixes.

4.1 Example Control Signal Enumeration

A sample of instructions and their control signals has been developed to demonstrate the inner workings of Overpowered Assembly Language .

	jr	regwrite	regswap	opsrc	ji	hcf	memread	memwrite	byte
`add Ra, #imm8`	0	1	0	1	0	0	0	0	0
`and Ra, #imm8`	0	1	0	1	0	0	0	0	0
`swp Ra`	0	1	1	X	0	0	0	0	0
`lda #imm16`	0	1	0	1	0	0	1	0	0
`sta #imm16`	0	0	0	1	0	0	0	1	0
`jne #imm16`	0	0	0	1	1	0	0	0	0
`sll Ra, #imm8`	0	1	0	1	0	0	0	0	0
`inv Ra`	0	1	0	1	0	0	0	0	0

4.2 Sample Program

A sample program, with commentary, has been developed as an example of the simplicity and powerful nature of Overpowered Assembly Language.

```
// The first section of code should set up the environment
// (stack, entrypoint, interrupt vector table, etc.).
    ldi     stack     // set the stack pointer
    swp     SP        // set the stack pointer
    jap     Main      // jump to entrypoint
    hcf               // halt and catch fire



// constants
stack_size byte 64              // byte - assembler directive: sets the
                                // value of a byte
array_element_size byte 1       // word - assembler directive: sets the
                                // value of a word
array_length byte 12
    align



// variables
    save stack_size       // save - assembler directive: allocates space
                          // for the given number of bytes
stack save 2
array save (array_element_size * array_length)
    align



// Program entrypoint.
// Calls the FibFill subroutine.
Main
    sub     SP,#6
    swp     SP
    ldi     #array_element_size
    ldab    [A,#0]
    sta     [SP,#5]
    ldi     #array_length
    ldab    [A,#0]
    sta     [SP,#4]
    ldi     #array
    sta     [SP,#2]
    jap     FibFill
    swp     SP
    add     #6
    swp     SP
    hcf



// FillFib
```

```
//
// Fills an array of given address, length, and element size with the
// fibonacci sequence, and returns the last number calculated.
//
// This code assumes that the array length is at least 3.
// Error checking has not yet been implemented.
//
// Parameters:
//    M[SP + 5] - (u1) array length
//    M[SP + 4] - (u1) array element size
//    M[SP + 2] - (u2) array address
//    M[SP]     - (u2) return address
// Returns:
//    A         - (u1) the last fibonacci number calculated
FibFill
    sub     SP,#6
    swp     X
    sta     [SP,#0]
    swp     Y
    sta     [SP,#2]
    ldab    [SP,#(5 + 6)]
    sub     #2
    stab    [SP,#4]
    lda     [SP,#(2 + 6)]
    swp     Y
    lda     [SP,#(4 + 6)]
    cmp     #1
    jeq     FibFill_Byte
FibFill_Word
    ldib    #0          // set the first two elements to 0 and 1
    sta     [Y,#0]
    inc
    sta     [Y,#2]
    swp     X
FibFill_Word_Loop
    lda     [Y,#0]      // Y holds the index of the second to last stored number
    add     X           // X should hold the last stored number
    sta     [Y,#4]
    swp     X
    add     Y,#2
    swp     Y
    ldab    [SP,#4]
    dec
    stab    [SP,#4]
    jzs     FibFill_Loop_Word
    jmp     FibFill_Done
FibFill_Byte
    ldib    #0
    stab    [Y,#0]
    inc
    stab    [Y,#1]
FibFill_Byte_Loop
    ldab    [Y,#0]      // Y holds the index of the second to last stored number
    add     X           // X should hold the last stored number
    stab    [Y,#2]
    swp     X
    add     Y,#1
    swp     Y
```

```
    ldab    [SP,#4]
    dec
    stab    [SP,#4]
    jzs     FibFill_Loop_Byte
FibFill_Done
    lda     [SP,#2]    // restore registers, the result should be in X
    swp     Y
    lda     [SP,#0]
    swp     X          // the result is now in A
    swp     SP         // balance the stack
    add     #6
    swp     SP
    ret                // return
```

The same program is included below, with Register Transfer Notation for each operation.

```
    ldi     stack                   A <- stack
    swp     SP                      A <- SP; SP <- A
    jap     Main                    M[SP, 2] <- PC
                                    PC <- PC + SignExtend(Main - PC, 1)
    hcf                             PC <- PC - 2



stack_size byte 64
array_element_size byte 1
array_length byte 12
    align



    save stack_size
stack save 2
array save (array_element_size * array_length)
    align



Main
    sub     SP,#6                   A <- SP - 6
    swp     SP                      A <- SP; SP <- A
    ldib    #array_element_size     A <- ZeroExtend(array_element_size, 1)
    ldab    [A,#0]                  A <- ZeroExtend(M[A + 0, 1], 1)
    sta     [SP,#5]                 M[SP + 5, 2] <- A
    ldib    #array_length           A <- ZeroExtend(array_length, 1)
    ldab    [A,#0]                  A <- ZeroExtend(M[A + 0, 1], 1)
    sta     [SP,#4]                 M[SP + 4, 2] <- A
    ldib    #array                  A <- ZeroExtend(array, 1)
    sta     [SP,#2]                 M[SP + 2, 2] <- A
    jap     FibFill                 M[SP, 2] <- PC
                                    PC <- PC + SignExtend(FibFill - PC, 1)
    swp     SP                      A <- SP; SP <- A
    add     #6                      A <- A + 6
    swp     SP                      A <- SP; SP <- A
    hcf                             PC <- PC - 2



FibFill
    sub     SP,#6                   A <- SP - 6
    swp     X                       A <- X; X <- A
    sta     [SP,#0]                 M[SP + 0, 2] <- A
    swp     Y                       A <- Y; Y <- A
    sta     [SP,#2]                 M[SP + 2, 2] <- A
    ldab    [SP,#(5 + 6)]           A <- ZeroExtend(M[SP + 11, 1], 1)
    sub     #2                      A <- A - 2
    stab    [SP,#4]                 M[SP + 4, 1] <- A
    lda     [SP,#(2 + 6)]           A <- M[SP + 8, 2]
    swp     Y                       A <- Y; Y <- A
    lda     [SP,#(4 + 6)]           A <- M[SP + 10, 2]
```

```
    cmp     #1                  A - 1
    jeq     FibFill_Byte        PC <- PC
                                + SignExtend(FibFill_Byte - PC, 1)
FibFill_Word
    ldib    #0                  A <- ZeroExtend(0, 1)
    sta     [Y,#0]              M[Y + 0, 2] <- A
    inc                         A <- A + 1
    sta     [Y,#2]              M[Y + 2, 2] <- A
    swp     X                   A <- X; X <- A
FibFill_Word_Loop
    lda     [Y,#0]              A <- M[Y + 0, 2]
    add     X                   A <- A + X
    sta     [Y,#4]              M[Y + 4, 2] <- A
    swp     X                   A <- X; X <- A
    add     Y,#2                A <- Y + 2
    swp     Y                   A <- Y; Y <- A
    ldab    [SP,#4]             A <- ZeroExtend(M[SP + 4, 1], 1)
    dec                         A <- A - 1
    stab    [SP,#4]             M[SP + 4, 1] <- A
    jzs     FibFill_Loop_Word   PC <- PC
                                + SignExtend(FibFill_Loop_Word - PC, 1)
    jmp     FibFill_Done        PC <- PC
                                + SignExtend(FibFill_Done - PC, 1)
FibFill_Byte
    ldib    #0                  A <- ZeroExtend(0, 1)
    stab    [Y,#0]              M[Y + 0, 1] <- A
    inc                         A <- A + 1
    stab    [Y,#1]              M[Y + 1, 1] <- A
FibFill_Byte_Loop
    ldab    [Y,#0]              A <- ZeroExtend(M[Y + 0, 1], 1)
    add     X                   A <- A + X
    stab    [Y,#2]              M[Y + 2, 1] <- A
    swp     X                   A <- X; X <- A
    add     Y,#1                A <- Y + 1
    swp     Y                   A <- Y; Y <- A
    ldab    [SP,#4]             A <- ZeroExtend(M[SP + 4, 1], 1)
    dec                         A <- A - 1
    stab    [SP,#4]             M[SP + 4, 1] <- A
    jzs     FibFill_Loop_Byte   PC <- PC
                                + SignExtend(FibFill_Loop_Byte - PC, 1)
FibFill_Done
    lda     [SP,#2]             A <- M[SP + 2, 2]
    swp     Y                   A <- Y; Y <- A
    lda     [SP,#0]             A <- M[SP + 0, 2]
    swp     X                   A <- X; X <- A
    swp     SP                  A <- SP; SP <- A
    add     #6                  A <- A + 6
    swp     SP                  A <- SP; SP <- A
    ret                         PC <- M[SP + 2, 2]
```