WUT-4 Assembly Language

Overview

This is the WUT-4 Assembly Language documentation.unlike most Assembly Language documents the op codes of the Assembly Language are not specified here. For the op codes see the WUT-4 architecture document.

The Assembly Language is line-oriented. Each line consists of an optional label, an opcode or directive, arguments (that may be optional), and a comment field that is always optional. Arguments are separated from each other by spaces or optional commas. if commas are present, they are treated exactly like spaces.The optional comment field, if present, must be preceded by a semicolon. A label, if present, must be followed by a colon.In fact, all fields are optional because blank lines are legal and so are lines consisting only the semicolon followed by a comment.

Thus a typical line might appear:

```
label:        mnemonic argument argument, argument      ; some comment
```

or

```
; This is a standalone comment
```

Labels may be forward referenced.

Stylistically it is preferable to either use commas consistently or avoid their use. but the assembler does not enforce this. Although the documentation sometimes defines mnemonics in uppercase, the assembler accepts instruction mnemonics only in lower case.

As it assembles the file, the assembly maintains the value of dollar which is the next location to be filled in the output file. The assembler must actually maintain two values of dollar, one for the code segment and one for the data segment. The directives **`.code`** and **`.data`** are used to make one of the values of dollar current. Both values begin at 0. Values of dollar become virtual addresses in their respective segments.When a label appears it is assigned the value of dollar. In constant expressions (defined below), the value of dollar and of previously-defined labels may be referenced. The current value of dollar is referenced using a literal dollar sign. Thus, it is legal to write the following:

```
mystring:  .byte "some string\n\0"
.set mystringlen $-mystring
```

After which the value of **`mystringlen`** should be 12

The language supports constant expressions which may consist of numeric values, the operators +, -, *, /, (truncating integer division), labels, and symbols previously defined set with the .set directive (for the complete definition, see below).

Labels may be defined before they are used. but symbols created with **`.set`** must be defined before use. Labels Used in flow control instructions may be forward referenced, but labels used in constant expressions must be defined before use in lexical order.

Output file format

The output file consists of a 16 byte header followed by the code segment words and then followed by the data segment (initialized data) if any. the header consists of a structure containing three 16-bit unsigned integers and 10 bytes (5 16-bit words) of reserved space. The first 16-bit unsigned integer of the file contains the value 0xDDD1. This is the “magic number” helpful for identifying WUT-4 executables. The second unsigned integer contains the number of file bytes that should be loaded in the code section. The third unsigned integer contains the number of file bytes that should be loaded in the data segment, Data segment bytes immediately follow code segment bytes in the file.Although instructions in the code space must be aligned on 16-bit boundaries, no assumptions can be made about the size of the entire segment because it is perfectly legal to allocate static data in the code segment. So, for example, the code segment may end with static string data having an odd length.

So if we imagine this pseudocode for the header:

```
struct header{
 magic: u16
 codesize:u16
 datasize:u16
 reserved: [5] u16
}
```

Then the file offset of the first byte of the code segment, if any, would be **`sizeof (header)`**. The file offset of the first byte of the data segment would be **`sizeof (header)+header.codesize`**.

Symbols

Labels and the values of. set directives are examples of symbols in the assembler. Symbols must begin with the letter, may contain letters, numbers or underscores, and have a maximum

length of 16 characters. Assembler source code is written in ASCII. Quoted strings may contain utf-8.

Labels

Labels are symbols followed by a colon that appear in the first field of a source line. Labels assume the current value of dollar when they appear.

Directives

The assembly language supports the following directives.

Name	**Arguments**	**Effect**
.align	one number. unit is bytes	Adjust the position of dollar for the current segment to a multiple of bytes given by the argument. The value may be any constant expression.
.bytes	list of byte values, or a quoted string.If a quoted string the data is taken as utf-8 encoded.Such strings are not by default null terminated. Use \0 where required.	Place the bytes successively in the current segment at dollar, incrementing dollar. Each 8-bit value may be any constant expression. Numeric values may range from -128 to 255. Values which do not fit in 8 Bits are truncated with a warning.
.words	list of 16-bit word values	Each 16-bit value may be any constant expression.
.space	one number. Unit is bytes.	Increment dollar by the argument, which may be any constant expression.
.code	none	Make the code segment the current segment.
.data	none	Make the data segment the current segment
.set	symbol name , value	Define the symbol to represent the value. All such defined symbols are constants and may

		not be redefined.Their value may be any constant expression. Constant expressions are defined below. Unlike labels which may be forward-referenced, Symbols defined with **`.set`** must be defined before use in lexical order.

In strings that follow **`.bytes`**, utf-8 characters (including multibyte characters) may occur literally. Also, the following Escape sequences are defined with the same meanings as in the C language: \0 \n \r \b \t. In addition, within any string, a backslash maybe followed by two hexadecimal characters with letters in either upper or lowercase to produce the corresponding byte.This allows the entry of utf-8 sequences, albeit clumsily. Aside from these cases in quoted strings, all assembler source code must be ASCII.

Constant Expressions

Expressions may appear in .set directives or as the values of immediate operands. These expressions are composed of numbers and operators with the usual meanings. Labels may appear in constant expressions if their value is known; in other words labels may not be forward referenced in constant expressions. The value of dollar (“$”) may be used. The operators include +, binary -, *, / for truncating integer division, << and >> for logical shifts, unary -, bitwise &, |, ~ for bitwise not. Unary minus and bitwise not have the highest precedence, followed by the multiplicative operators: * / & << >> leaving the additive operators + binary - and bitwise | with the lowest precedence. So there are three levels of precedence; within each level . All operators associate tightly to the left. Parens () may be used where required to control precedence or association. Numeric values may appear in decimal, octal, or hexadecimal when prefixed with nothing at all, 0o, or 0x respectively. In addition, binary values prefixed with 0b, containing 1’s and 0’s., may appear.Numbers may contain underscores to separate digits strings. The underscores have no significance to the value.

Assembler Mnemonics

Many instructions have an argument that is an immediate value. the immediate value if present is always the last argument to the instruction.Immediate values are always optional and are assumed to be zero if not given.

All the mnemonics and their bit layouts are defined in the accompanying architecture document.

Assembler Pseudoinstructions

These are assembly supported mnemonics that expand to multiple instruction op codes(mnemonics). They might not appear in disassemblies because a disassembler may simply produce the actual opcodes. for example, the pseudo ldi may appear in a disassembly as an lui instruction followed by an adi instruction.

LDW r0, r0+0 (which would assemble to 0x0000) generates an assembler error. The opcode 0x0000, if executed, generates an illegal instruction trap. This is a special case intended to catch execution in uninitialized code space memory. The DIE mnemonic generates 0xFFFF. It should be used for aborts.

Pseudoinstructions supported by the assembler that expand to other instructions include:

ldi rT, imm16 (load immediate with a value in 0..0xFFFF) is a nontrivial builtin assembler pseudo:

If imm16 < 0x40, it expands to ADI rT, r0, imm6 (“add short immediate”, ADI) which moves the small value to the target register.

if imm16 & 0xFFC0 == imm16, it expands to LUI rT, imm10 (“load upper immediate”, LUI)

Otherwise, it expands to (LUI rT, (imm16&0xFFC0)>>6 ; ADI rT, r0, imm16&0x3F) (“load full size immediate”).

If rT is 0, the LINK register is the target.

The assembler should optimize use of ldi to the shortest form that will produce the desired result

The opcodes generated to move small immediate values to registers should be just adi instructions

The Lui instructions are unnecessary for immediate values that fit in a 7-bit signed binary field.

jal label

Shorthand for jal link (0), label

jal rT, label

Expands to lui rT, (label&0xFFC0)>>6 ; jal rT, rT, (label&0x3F).

rT may be LINK(0).

jal rT, rS, label

Expands to lui rT, (label&0xFFC0)>>6 ; jal rT, rS, (label&0x3F).

rT, rS, or both may be LINK.

The rA field holds rT (the PC save register). The rB field holds rS.

This fully general form subsumes all the shorthands.

mv rT, rS

Expands to adi rT, rS, 0
moves (assigns) the value in rS to RT

srr rA, rB, imm7 (LI rB, imm7 ; LSP rA, rB) = li ; srr - “Special Register Read”
Load rA from one of the 128 special registers at imm7 using rB.
Special register space is word addressable.

srw rA, rB, imm7 (LI rB, imm7 ; SSP rA, rB) = li ; srw - “Special Register Write”
Store rA to one of the 128 special registers at imm7 using rB.
Special register space is word addressable.

ret [rN] - return
Implemented as JI rN. Rn may be LINK (0).
If no argument is given, rN is LINK. rN is modified.

sla rN, sll rN - shift left arithmetic / logical
SLA is implemented as ADC rN, rN, rN; SLL is implemented as ADD rN, rN, rN. Unlike SRA and SRL,
these two instructions do not set the carry flag to the “lost” bit–rather they set it like an add (since they are implemented as adds.).