CASPR - A Configurable Assembler Program
Documentation of Porting and Use

1 Background

Caspr was designed to be a generic and portable assembler for simple processor architectures. For processors which do not already have a dedicated assembler, caspr avoids the need for hand assembly, which is tedious and prone to errors. Caspr can use most small instruction sets by means of an appropriate configuration file. This configuration file is then read at assembler run time, and used to interpret a given assembly program using that language. This approach has the substantial advantage of separating the user from the internal operation of the assembler, and only requires that the user know about the target instruction set.

2 Instruction Set Configuration

To implement a given processor instruction set, one must know all the instructions that will need to be implemented, the corresponding opcodes for those instructions, and the size and positioning of whatever instruction arguments are used. To simplify, each instruction will be assembled into binary machine codes, multiples of eight bits each (byte alignment for standard memory systems). Within the bits of this binary output, a certain set of bits represent the instruction opcode used to identify the instruction, while other bits represent the arguments used by that instruction. Thus, each instruction has a unique generation pattern, specifying bits required for an opcode, and any number of integer values, which are then inserted into the final machine code at set locations.

To clarify with an example, suppose on a hypothetical machine, an add operation has an opcode of “0101” and takes three register arguments, each of which specify one of 16 registers. Thus, out of this two-byte instruction, the first byte would consist of the bits “0101” followed by a four bit number specifying one of sixteen registers (possible values 0 to 15). Similarly, the second byte would consist of of a pair of four bit fields, specifying the second and third registers. Again, these values range from 0 to 15.

To instruct the assembler to parse and generate correct code for this instruction, we would need to write the line below, within a file named “machine.cfg”.

```
add 4 4 4 { 0101 (0) (1) (2) }
```

Note that the structure of this line specifies first the assembly mnemonic, and then three fours, each representing a four bit number argument to use in the assembly process. The output pattern is then specified between a pair of curly braces, as four constant bits followed by three numbers surrounded in parentheses. Each of these values in parentheses represent the bit representation of one of the integer arguments (with 0 representing the first argument). Within the output pattern, an instruction may consist of any combination of set opcode bits and argument specifiers (repeated if desired). Any whitespace inside the pattern is ignored, and can be used for separating or lining fields up visually for organization and clarity. The only restriction here is that the final output must be a multiple of 8 bits, to make the output byte aligned and compatible with standard memories.

3 Tiny CPU - A Full Assembler Configuration

Tiny is a small processor architecture, which can be used to demonstrate porting caspr to a new architecture. At its simplest, Tiny is a accumulator machine with only five instructions. Instructions are either one or two bytes, and always contain the opcode within the most significant 3 bits of the first byte. The entire instruction set is represented below.

Assembly	Operation	Instruction Format	Opcode	Z	Cycles
ADD $M	Acc <- Acc + M	Op(3) unused M(8)	001	Y	5
STR $M	M <- Acc	Op(3) unused M(8)	010	N	4
CLA	Acc <- 0	Op(3) unused	011	N	1
JNZ $M	PC <- M if Z=0	Op(3) unused M(8)	101	N	3
RST	PC <- 0	Op(3) unused	111	N	1

For the most part, the assembler is interested only in the assembly mnemonic, its argument (if any), and the instruction format including the instruction opcode. Aside from programmer comments for future maintainability, this becomes a direct translation into a syntax that caspr will understand. Compare the above table with the below configuration for this language.

```
add 8 { 001 00000 (0) }
str 8 { 010 00000 (0) }
cla   { 011 00000     }
jnz 8 { 101 00000 (0) }
rst   { 111 00000     }
```

As mentioned before, caspr ignores whitespace and is case insensitive. This allows configuration files to be set up in a way to aid in readability, and identify the values in important bit fields at a glance. For example, the JNZ instruction consists of an opcode of “101” followed by five unused bits (all set to 0 in this example) in the first byte followed by the 8 bit argument taking the space within the second byte. To check that the instruction is indeed a multiple of eight bits (byte aligned), we can look to see that the instruction is made up of a three bit opcode field, a five bit unused/reserved field, and argument number 0, which is marked next to the assembly mnemonic as being eight bits wide. Thus, this instruction totals sixteen bits, and therefore takes an even two bytes inside processor memory.

3.1 Full Listing of tiny.cfg

```
; Architecture file for the Tiny CPU

; Output will be an Intel MIF file format
.outfmt mif

; MIF dimentions (128 words, each 8 bits wide)
.mifwords 128
.mifwidth 8

; Opcodes for core instruction set
add  8  { 001 00000 (0) } ; Add
str  8  { 010 00000 (0) } ; Store
cla     { 011 00000     } ; Clear Accumulator
jnz  8  { 101 00000 (0) } ; Jump if not zero
rst     { 111 00000     } ; Reset

byte 8  { (0)           } ; One byte constant
```

Note in the above full example the use of assembler comments starting with the semicolon character, and assembler directives starting with a period. All characters after a semicolon are completely ignored, while strings starting with periods are given special treatment, and are used to modify how the assembler generates output. The directives used in this configuration specify the appropriate dimensions of the MIF format required by the target architecture. A full listing of these directives is provided in the appendix.

Additionally, as this instruction set has no way to address immediate data, it may be useful to have a way to set arbitrary bytes within the program ROM. To facilitate this, this example lists another pseudo-instruction ("byte"). This mnemonic is not associated with any given machine code, so the bits it will set are from the argument we pass to it. Thus, to set an arbitrary byte, we pass an 8 bit value as an argument, and generate a pseudo instruction containing just that data. Combined with the appropriate directives, this allows a great deal of control to allow placing arbitrary constants within program ROM. An example of this follows in the next section.

Also note that by modifying the 8 bit "byte" mnemonic, we could create a similar instruction to to set a 16 bit word instead, but this will be left as an exercise to the reader.

4 An Example Tiny CPU Assembly Program

Below, you will see a simple assembly program written for the Tiny CPU instruction set. Similar to most assembly language conventions, comments are preceeded by semicolons, and line labels are followed by colons. Numbers are interpreted as hexadecimal if prefixed by "0x", as octal if prefixed only by "0", and decimal otherwise. In the below example, a shorthand notation of using a "$", which indicates that the number should be parsed as if it had a "0x" prefix.

4.1 Full Listing of tiny1.asm

```
.arch tiny
.outfmt rom
.define OUT $FF

        cla                     ; Clear Acc and then adding the operand is
LOOP:   add ONE                 ; equivalent to move the operand to Acc
        str OUT
        jnz LOOP
        rst

        ;; begin writing data to ROM constant section
        .org $78

ONE:    byte $01
```

Note the use of the .arch directive on the first line. It specifies which architecture configuration to use for assembling this program. This is required as caspr does not include this information in its own compilation, but also allows caspr to be used a single assembler for many architectures. By including this information, caspr will search for a file named "tiny.cfg" and parse it for assembler information.

Also used is the .outfmt directive. Remember that this is also included in the Tiny configuration file. However, as the directive in the assembly program is parsed after the architecture configuration (".arch tiny"), the use within the program will override the value set in the global architecture configuration. This allows settings be set on a default basis, and changed for special cases as needed.

Another useful directive is .define, which allows the programmer to assign arbitrary values to symbolic names, and refer to them later in the assembly file. Such program constants are useful for referring to memory mapped I/O addresses or limits set at assembly time.

Lastly, to set aside a separate ROM section for setting constants, this program uses the .org directive, which will set a new location in program ROM, for instruction assembly. Any valid instruction mnemonics following this directive, will start at this new address location. In this case, the byte mnemonic would appear in program ROM at address 0x78, and any mnemonic following would appear at addres 0x79. This is a simple way to place constants at known locations.

4.2 Running the Assembler

Enter the above program into a file named tiny1.asm, and the architecture configuration into a file named tiny.cfg. Open a text console (such as xterm), change to the directory with your files and run the caspr with your assembly program as the argument. You screen should look like the following:

```
> ./caspr tiny1.asm
Output name is 'tiny1.rom'
```

Viewing the output file shows the following:

```
> cat tiny1.rom
0x0000 | 60 20 78 40 FF A0 01 E0
0x0008 | 00 00 00 00 00 00 00 00
0x0010 | 00 00 00 00 00 00 00 00
0x0018 | 00 00 00 00 00 00 00 00
0x0020 | 00 00 00 00 00 00 00 00
0x0028 | 00 00 00 00 00 00 00 00
0x0030 | 00 00 00 00 00 00 00 00
0x0038 | 00 00 00 00 00 00 00 00
0x0040 | 00 00 00 00 00 00 00 00
0x0048 | 00 00 00 00 00 00 00 00
0x0050 | 00 00 00 00 00 00 00 00
0x0058 | 00 00 00 00 00 00 00 00
0x0060 | 00 00 00 00 00 00 00 00
0x0068 | 00 00 00 00 00 00 00 00
0x0070 | 00 00 00 00 00 00 00 00
0x0078 | 01 00 00 00 00 00 00 00
```

Even though the architecture configuration file we created specified that the output format should be an Intel MIF format, the assembly directive within our assembly program has changed it to an easilly readable text ROM format. Removing the appropriate .outfmt directive in the tiny1.asm file will change the output to the default MIF file on the next assemble.

However, since this output is easilly readable, the user can compare the expected byte output of each instruction to the results of the above compilation. Appropriate opcodes for each instruction should be set in the most significant three bits, and 8 bit arguments should be in the following byte. Also note that value 0x01 is correctly located at the position 0x78, and the add instruction using the value of the line label "ONE" references the appropriate address. This address can be modified by changing the value within the .org directive, and placed anywhere within the output program's address space.

4.3 Assembler Limitations

To keep the internal function of caspr relatively simple, it makes several assumptions about the structure of its input, and the boundaries of the output. Although most of these should not cause concern for small 8-bit and 16-bit processors, they are included here for completeness. In future versions, these limitations should be removed.

- A hard limit of 16 numeric arguments for any assembly instruction/mnemonic
- A maximum of 64 characters for any assembly mnemonic, line labels, or numeric strings
- A maximum of 512 characters for any file path
- A maximum of 32 bits for any generated instruction

5 Assignments and Exercises

- Append configuration information to tiny.cfg for the three extra instructions created in Lab 2. This will create a complete configuration for your modified version of the processor.
- Test your new configuration by assembling a few of the programs from previous labs, comparing them to the expected output.
- Generate a configuration file for your own project processor, and verify that instructions are assembled correctly on a few small test programs.

A Listing of All Caspr Directives

.arch NAME Instruct caspr to locate a file named “NAME.cfg” and use it for assembly mnemonic information. Note this filename must be lowercase, due to the way caspr handles upper and lowercase string insensitivity.

.define NAME VALUE Create a named value which which be interpreted as a numeric value within assembly instruction arguments.

.mifwidth VALUE Forces generated MIF files to have an internal data word width of VALUE bits (VALUE must be a multiple of 8). Useful for architectures which read data two or more bytes at a time. Only valid when outputting as a MIF file.

.mifwords VALUE Forces generated MIF files to contain a total of VALUE words. Only valid when outputting as a MIF file.

.org ADDR Modifies the internal ROM program offset counter, and forces the next instruction mnemonic to be placed at ADDR (must be a numeric constant).

.outfmt FMT Output in either Intel MIF format (“mif”) or a readable text based ROM (“rom”). Other strings are invalid.