Cdm8 extended Programmer’s manual

Version 0.1

Merzlyakov Ilya, Korotkov Nikita, Odina Anastasia

May 30, 2022

Contents

1 Cdm8e architecture

1.1 Memory

Cdm8e has 8-bit data bus and 16-bit address bus. Cdm8e can address 64 kilobytes of memory, but only the first 256 bytes of address space can be used to store data. All address space can be used to store program.

1.1.1 Von Neuman and Harvard architectures

Cdm8e can be used in systems with Von Neuman (same address space for instructions and data) and Harvard (different address spaces for instructions and data) architectures.

For this purpose, cdm8e has *data/ins'* output, which is low when processor accesses data memory.

However, our development tools (emulator and debugger) only support Harvard architecture.

1.2 Registers

Cdm8e has 4 8-bit general purpose registers (r0, r1, r2, r3), 8-bit stack pointer register (SP), 8-bit processor status register (PS) and 16-bit program counter register (PC). All registers are initialized to zero upon reset.

1.2.1 General purpose registers

These registers can be used as operands for instructions that accept *register* operand.

1.2.2 Stack pointer

The Stack Pointer is an 8-bit register that holds a pointer to the top of the stack.
The stack is a data structure of variable size made up of memory cells. The first byte of the stack is held at memory location `0xFF`, and the stack grows *down* memory from there. It is managed using SP register, which contains the address of the most recent byte stored on the stack. It is the responsibility of the programmer to manage the stack properly. Each *push* instruction makes the stack **grow** in size by 1 byte, causing it to get closer and closer to those locations where program instructions and initial data are stored, so too many `push`es without a `pop` can cause a program to be corrupted by being over-written by the stack.

1.2.3 Program counter

The Program Counter is an 16-bit register that holds the address of the next machine instruction that will be executed. The PC is therefore used as a pointer to the start address of the next instruction. When an uninterrupted sequence of instructions is being executed PC is updated so that it points at the next memory location. Each instruction occupies a known number of bytes, and when a byte is fetched the PC is immediately incremented to point to the next byte before anything else happens, in particular before the instruction starts to be executed. The PC cannot be accessed directly by a programmer, but its contents can be updated indirectly by control instructions.

1.2.4 Processor status

The Processor Status register holds an 8-bit bitstring, where ech bit have it's own function.

bit number	7	6	5	4	3	2	1	0
function	I	R	R	R	C	V	Z	N

- **I** is Interrups enabled flag. When it is set, processor will react to external interrupts
- **C** is the Carry flag. This bit will be updated on completion of any arithmetic operation that produces a carry-out from bit 7 of the result. The carry-out value (whether 0 or 1) is transferred to the C bit of the PS register.
- **V** is the oVerflow flag. This bit will be updated on completion of any arithmetic operation that can give rise to a two's complement overflow event. The flag is set if an overflow event has been detected and cleared otherwise.
- **Z** is the Zero flag. This bit will be updated on completion of any arithmetic or logic operation. The flag is set when the result is an all-zeros bit-string and cleared when it is not.
- **N** is the Negative flag. This bit will be updated on completion of any arithmetic or logic operation. It is set if the operation resulted in a bit-string that is, or would be interpreted as a negative number based on the sign bit. Specifically, the value of bit 7 is copied to the N flag, so N is set when the result is a bit-string that could represent a negative number and cleared otherwise.
- **R** are reserved bits. These bits are unused.

The PS register is examined by control instructions to change the PC (thus changing which instruction will be executed next) if a certain combination of flag values is present at this point. It is important to understand that the PS register is updated based on the category of operation performed (arithmetic/logic operation), not because the update is wanted by the program. If the result of an operation is to be used to control program execution the flags must be checked immediately, by the next machine instruction, unless the intervening instructions are such that they do not affect the flags.

1.3 Interrupts

Interrupts are external events that require immediate reaction from program. When interrupts are enabled and an interrupt occurs, processor pushes current value of PC and PSR onto stack and loads Interrupt Service Routine (ISR) address and new value of PS from from interrupt vector, thus passing control to ISR. To return from ISR and restore normal program execution, *rti* instruction must be used.

Cdm8e has 8 interrupt vectors which form interrupt vector table that must be placed in code memory at address 0xFFE0. Each interrupt vector has length of 4 bytes and has following structure: **<lower byte of ISR address><higher byte of ISR address><new value of PS><padding byte (ignored)>**

2 Cdm8e assembly

Cdm8e assembler syntax is mostly identical to syntax of *cocas.py*, which is described in *tome.pdf*. However, it has some new features which will be described here.

For more detailed syntax description see assembler grammar.

2.1 Goto

The *goto* statement should be used instead of branch instructions, because it is can handle long jumps. If the jump is short, assembler replaces *goto* with corresponding branch instruction, otherwise it is replaced with a pair of branch and jmp instructions.

The syntax is `goto <branch mnemonic>, const`

Example:

```
asect 0x00
goto z, near_code       # equivalent to bz near_code, takes 2 byes
goto hs, far_code       # will branch on 'higher or same', takes 5 bytes

near_code:
ldi r0, 1

asect 0x1337
far_code:
ldi r0, 2
end
```

2.2 Address arithmetics

All instructions that accept integer literals and labels now accept address expressions. In address expressions it is possible to add and subtract numbers, labels and tplate fields. To get low and or high byte of an expression, `low(expression)` and `high(expression)` specifiers can be used.
Examples:

```
asect 0x00

ldi r0, low(sting_end - string) # load string length into r0

ldi r0, high(far_code)          # push 16 bit address onto stack
push r0
ldi r0, low(far_code)
push r0


string: dc "Hello, world"
string_end:

asect 0x1337
far_code:

end
```

3 List of cdm8e instructions

3.1 Memory access instructions

`ldi rn, const` *const* → r*n* Flags unchanged

Load the **immediate** single-byte data item *const into* r*n. The bit-string representing* const is copied into r*n*.

2-byte machine code instruction
Opcode: `110100`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: immediate value (second byte of instruction)

Example usage
Instruction: `ldi r1, 0x6E`
In binary: `11010001 01101110`
Before: `N/A`
After: `r1 contains 01101110`

`ld rn,rm` *r*n → r*m* Flags unchanged

Load a byte into r*m* from the data memory cell addressed by r*n*.
(**r*n* is the memory cell *pointed to* by r*n*. The bit-string read from this memory cell is copied into r*m*.)

1-byte machine code instruction
Opcode: `1011`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

Example usage
Instruction: `ld r0, r3`
In binary: `10110011`
Before: `r0 contains 01111001`
After: `r0 and mem[01111001] unchanged, r3 contains a copy of mem[01111001]`

`ldc rn,rm` *rn → rm Flags unchanged

Load a byte into r*m* from the code memory cell addressed by r*n*.
(*r*n* is the memory cell *pointed to* by r*n*. The bit-string read from this memory cell is copied into r*m*.)

1-byte machine code instruction
Opcode: 1111
Operand1: 2-bit register number (00, 01, 10 or 11)
Operand2: 2-bit register number (00, 01, 10 or 11)

`st rn,rm` rm → *rn Flags unchanged

Store the byte in r*m* to the memory cell addressed by r*n*.
(*r*n* is the memory cell *pointed to* by r*n*. This cell is over-written by the bit-string copied from r*m*.)

1-byte machine code instruction
Opcode: 1010
Operand1: 2-bit register number (00, 01, 10 or 11)
Operand2: 2-bit register number (00, 01, 10 or 11)

Example usage
Instruction: st r1, r0
In binary: 10100100
Before: r1 contains 00000110
After: r1 and r0 unchanged, mem[00000110] contains a copy of r0

3.2 Stack instructions

`push rn` ((SP-1)→SP) then (`rn` → *SP) Flags unchanged

Push the byte in `rn` onto the stack. SP is the *stack pointer* register. This is decremented, then used to point at a memory cell which is over-written by the bit-string copied from `rn`. (*SP is the memory cell pointed to by SP) **1-byte** machine code instruction

Opcode: `110000`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

Example usage
Instruction: `push r2`
In binary: `11000010`
Before: `SP contains 00000000`
After: `r2 unchanged, SP contains 11111111, mem[11111111] contains a copy of r2`

`pop rn` (*SP → `rn`) then ((SP+1)→SP) Flags unchanged

Pop a byte off the stack into `rn`. SP is the *stack pointer* register. This is used to point at a memory cell which is copied into `rn`, then incremented. (*SP is the memory cell pointed to by SP)

1-byte machine code instruction
Opcode: `110001`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

Example usage
Instruction: `pop r3`
In binary: `11000111`
Before: `SP contains 11111111`
After: `mem[11111111] unchanged, SP contains 00000000, r3 contains a copy of mem[11111111]`

`ldsa rn,const` (SP + *const*) → `rn` Flags unchanged

Load stack address + *const* into `rn`
This can be used in conjunction with *tplate*s to get address of local variable

2-byte machine code instruction
Opcode: `110010`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: immediate value (second byte of instruction)

`addsp` *const*	(SP + *const*) → SP	Flags unchanged

Add const to **Stack Pointer** This can be used in conjunction with tplates to push and pop tplates to and from stack.

2-byte machine code instruction
Opcode: `11001100`
Operand1: immediate value (second byte of instruction)
Operand2: None

`setsp` *const*	*const* → SP	Flags unchanged

Set Stack Pointer to *const*

2-byte machine code instruction
Opcode: `11001101`
Operand1: immediate value (second byte of instruction)
Operand2: None

`pushall`	r3 → *(SP-1); r2 → *(SP-2); r1 → *(SP-3); r0 → *(SP-4); (SP - 4) → SP	Flags unchanged

Push all registers onto stack

1-byte machine code instruction
Opcode: `11001110`
Operand1: None
Operand2: None

`popall`	*SP → r0; *(SP + 1) → r1; *(SP + 2) → r2; *(SP + 3) → r3; (SP + 4) → SP	Flags unchanged

Pop all registers from stack

1-byte machine code instruction
Opcode: `11001111`
Operand1: None
Operand2: None

3.3 Arithmetic instructions

Like all other Cdm8e operations these may be used on any bit-strings. However, they are *named* for the results they give when those bit-strings represent numbers.
The flags in the Processor Status (PS) register are affected by each of these operations. C and V are modified in the course of *calculating* the result, whereas Z and N depend solely on

the result bit-string: Z is 1 when the result is an all-zeros bit-pattern, and 0 otherwise, N is equal to bit 7 (the *sign bit*) of the result.
Conventionally, C is taken to be the value that is *carried out* from Column 7 of the bit-string, and V tells us whether there has been a *two's complement overflow* (e.g. when the result of adding together two bit-strings representing positive numbers in two's complement form is a bit-string that represents a negative number in two's complement form, such as 01000000 + 01100000 = 10100000).
It is important to remember, however, that the true 'meaning' of each of the status flags depends upon what the bit-strings being manipulated actually represent. For example, it is perfectly possible to apply an *add* operation to a pair of registers containing bit-strings that represent ASCII characters. Neither the resulting bit-string nor the flags would be terribly meaningful under such circumstances, and to interpret V=1 as a two's complement overflow (for example) would be pretty daft.

`move rn,rm`	$rm \rightarrow rn$	Z, N reflect result C, V become 0

Move `rn` **to** `rm`
Copies the content of `rn` to `rm`. `C` and `V` are cleared. `N` and `Z` are based on the modified `rn`.

2-byte machine code instruction
Opcode: `0000`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`add rn,rm`	$(rn + rm) \rightarrow rm$	C, V, Z, N reflect result

Add together the bit-strings in `rn` and `rm`, assuming they represent binary numbers.
The result is placed in `rm`.
C is the carry-out from column 7.
V is 1 when $rn_7 = rm_7$ before the operation and $rn_7 \neq rm_7$ afterwards. Otherwise V is 0.

1-byte machine code instruction
Opcode: `0001`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`addc rn,rm` $(rn + rm + C) \rightarrow rm$ C, V, Z, N reflect result

Add together the **C** flag (0 or 1) and the bit-strings in r*n* and r*m*, assuming they represent binary numbers.
The result is placed in r*m*.
Add-with-carry-in is used when performing *byte-sliced addition* on numbers that are represented by bit-strings made up of two or more bytes.
Beforehand the C flag holds a *carry-in* value (the carry-out from bit 7 of a lower-order byte), and afterwards its content is the *carry-out* from bit 7 of the addition.
V is 1 when $rn_7 = rm_7$ before the operation and $rn_7 \neq rm_7$ afterwards. Otherwise V is 0.

1-byte machine code instruction
Opcode: 0010
Operand1: 2-bit register number (00, 01, 10 or 11)
Operand2: 2-bit register number (00, 01, 10 or 11)

`sub rn,rm` $(rn - rm) \rightarrow rm$ C, V, Z, N reflect result

Subtract the byte in r*m* from the byte in r*n*, assuming they represent binary numbers.
The result is placed in r*m*.

1-byte machine code instruction
Opcode: 0011
Operand1: 2-bit register number (00, 01, 10 or 11)
Operand2: 2-bit register number (00, 01, 10 or 11)

`cmp rn,rm` Calculates $(rn - rm)$ C, V, Z, N reflect result

Compare r*m* with r*n*.
Assume the bytes in r*n* and r*m* represent binary numbers and perform the subtraction $(rn - rm)$.
Used to modify flags without affecting registers or memory.
The registers r*n* and r*m* remain unchanged by this operation. Any of the four flags may change.

1-byte machine code instruction
Opcode: 0111
Operand1: 2-bit register number (00, 01, 10 or 11)
Operand2: 2-bit register number (00, 01, 10 or 11)

`neg rn` $(-rn) \rightarrow rn$ C, V, Z, N reflect result

Negate `rn`
Replace the contents of `rn` by its 8-bit two's complement.
If `rn` holds the 8-bit two's complement representation of the numerical value *x* before the operation it will contain the 8-bit two's complement representation of $-x$ afterwards.[a]

1-byte machine code instruction
Opcode: `100001`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

[a]The exception to this is the number -128, represented by 10000000, which has 10000000 as its 8-bit two's complement. So negating -128 gives -128.

`inc rn` $(rn + 1) \rightarrow rn$ C, V, Z, N reflect result

Treats `rn` as a binary number, and adds 1 to it.
The addition 'wraps around', so when `rn` contains 11111111 beforehand it will contain 00000000 afterwards (and the C, V and Z flags will all be set to 1). The V flag will also be set to 1 by `inc` when 01111111 is incremented to 10000000, but otherwise it will be 0.

1-byte machine code instruction
Opcode: `100011`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`dec rn` $(rn - 1) \rightarrow rn$ C, V, Z, N reflect result

Treats `rn` as a binary number, and subtracts 1 from it.
The subtraction 'wraps around', so when `rn` contains 00000000 beforehand it will contain 11111111 afterwards (and the C, V and Z flags will all be set to 1). The only other time a flag will be set by `inc` is when 01111111 is incremented to 10000000 (in which case the V flag will be set to 1).

1-byte machine code instruction
Opcode: `100010`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`and rn,rm` (r*n* and r*m*) → r*m*

Z, N reflect result
C, V become 0

And r*n* with r*m*.
Computes the bitwise conjunction of r*n* and r*m* placing the result in r*m*.

2-byte machine code instruction
Opcode: `0100`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`or rn,rm` (r*n* or r*m*) → r*m*

Z, N reflect result
C, V become 0

Or r*n* with r*m*.
Computes the bitwise disjunction of r*n* and r*m* placing the result in r*m*.

2-byte machine code instruction
Opcode: `0101`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`xor rn,rm` (r*n* xor r*m*) → r*m*

Z, N reflect result
C, V become 0

Exclusive Or r*n* with r*m*.
Computes the bitwise exclusive-or of r*n* and r*m* placing the result in r*m*.

2-byte machine code instruction
Opcode: `0110`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: 2-bit register number (`00, 01, 10 or 11`)

`not rn` (not r*n*) → r*n*

Z, N reflect result
C, V become 0

Not r*n*.
Flips all bits in r*n*.

1-byte machine code instruction
Opcode: `100000`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`shra rn` (r*n* div 2) → r*n*

C, Z, N reflect result
V become 0

Arithmetic shift right r*n*
Shift every bit in the bit-string in r*n* one place to the right, whilst leaving the sign bit (bit 7) unchanged.
Bit 0 is shifted into C; V is 0; N & Z are based on the modified r*n*.
The effect on r*n* is the same as dividing a two's complement number by 2, with the result being that r*n* contains the *quotient* and C contains the *remainder* of the division.

1-byte machine code instruction
Opcode: `100110`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`shla rn` (r*n* × 2) → r*n*

C, V, Z, N reflect result
– become 0

Arithmetic shift left r*n*
Shift every bit in the bit-string in r*n* one place to the left, filling the least significant bit (bit 0) with 0.
Bit 7 is shifted into C; V is 1 if bit 7 changes and 0 if it does not; N & Z are based on the modified r*n*.
The effect on r*n* is the same as multiplying a two's complement number by 2.

1-byte machine code instruction
Opcode: `100101`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`shr rn` (r*n* ») → r*n*

C, Z, N reflect result
V become 0

Sliced shift right r*n*
Shifts the bit-string in r*n* one place to the right without maintaining the sign. The old value of C is shifted into the sign bit (bit 7), and bit 0 is shifted into C. V becomes 0. N and Z are based on the modified r*n*.

1-byte machine code instruction
Opcode: `100100`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

`rol rn`	(rotate-left `rn`) → `rn`	`C`, `V`, `Z`, `N` reflect result - become 0

Rotate left `rn`
Treats the bit-string in `rn` as if the opposite ends are directly connected, and shifts it left one place. The sign bit (b7) is shifted into b0, and also into `C`. `V` is cleared. `N` and `Z` are based on the modified `rn`.

1-byte machine code instruction
Opcode: `100111`
Operand1: 2-bit register number (`00, 01, 10 or 11`)
Operand2: None

3.4 Control instructions

`b* const`	*const* + PC → PC if codition matches	Flags unchanged

Branch (conditionally or unconditionally) to certain address.
Branches on cdm8e are relative, this means that *const* is treated as signed offset relative to current value of PC. Assembler automatically calculates difference between location of instruction and target value of PC, so user absolute value should be passed to this instruction in assembly. If the jump is too far, assembler will give an error. The * symbol should be replaced with one of the branch condition mnemonics, which will be described later in this document.

2-byte machine code instruction
Opcode: `1110`
Operand1: 4-bit branch condition code
Operand2: immediate value (second byte of instruction)

3.4.1 Branch conditions

op	name	test	interpretation
0	eq/z	Z	equal, equal to zero / Zero is set
1	ne/nz	¬Z	not equal, not zero, Zero is clear
2	hs/cs	C	unsigned higher or same/Carry is set
3	lo/cc	¬C	unsigned lower / Carry is clear
4	mi/npl	N	negative (minus)
5	pl/nmi	¬N	positive or zero (plus)
6	vs/nvc	V	oVerflow is set
7	vc/nvs	¬V	oVerflow is clear
8	hi/nlc	C ∧¬Z	unsigned higher
9	ls/nhi	¬C ∨ Z	unsigned lower or same
A	ge/nlt	(N∧V)∨(¬N∧¬V)	greater than or equal
B	lt/nge	N⊕V	less than, less than zero
C	gt/nle	(¬Z∧N∧V)∨(¬Z∧¬N∧¬V)	greater than,
D	le/ngt	(Z∨N∧¬V)∨(¬N∧V)	less than or equal
E	r/true	true	unconditional bRanch
F	false	false	no-op

`jmp` *const* — *const* → PC — Flags unchanged

Unconditional absolute **jump**. Pc is set to 16-bit const.

3-byte machine code instruction
Opcode: 11011101
Operand1: 2-byte immediate value in little-endian format
Operand2: None

`jsr` *const* — (SP-2) → SP; PC -> *SP; *const* → PC — Flags unchanged

Jump subroutine.
Branch unconditionally to a constant absolute address, pushing current value of PC to the stack. Pushed value is in the little endian format.

3-byte machine code instruction
Opcode: 11010110
Operand1: 2-byte immediate value in little-endian format
Operand2: None

`rts` *SP → PC; (SP+2) → SP Flags unchanged

Return from subroutine.
Pop PC value from stack.

1-byte machine code instruction
Opcode: `11010110`
Operand1: None
Operand2: None

`ioi`

Push cureent value of PC and PS onto stack, then load values od PC and SP from current interrupt vector. This instrcution executes instead of cureent instrcution if interrupt is triggeres. Software interrupt can be generated by using this instruction (but it is not possible to set interrupt vector with software).

1-byte machine code instruction
Opcode: `11011000`
Operand1: None
Operand2: None

`rti`

Return from **interrupt**.
Pop current value of PC and PS from stack. This instruction should meu sed at the end of interrupt handler to return to main program.

1-byte machine code instruction
Opcode: `11011001`
Operand1: None
Operand2: None

`crc` Flags unchanged

Exchange PC with value at the top of the stack.

1-byte machine code instruction
Opcode: `11011010`
Operand1: None
Operand2: None

3.5 Clock control instructions

`halt` Stop the clock — Flags unchanged

Halt the cdm8e processor.
Switches off the platform clock.
The PC is not updated, so if the clock is re-started the halt will be executed again. It makes no difference *how* the clock is re-started. **1-byte** machine code instruction

Opcode: `11010100`
Operand1: None
Operand2: None

`wait` Suspend the clock — Flags unchanged

Wait until an *interrupt* occurs.
Suspends the platform clock in anticipation of an interrupt.
The PC is not updated. If the clock is re-started the wait will be executed again *unless* the re-start is initiated by a hardware interrupt, in which case the PC is loaded with the start address of an interrupt service routine, and *then* the clock is re-started.

1-byte machine code instruction
Opcode: `11010101`
Operand1: None
Operand2: None

3.6 Macros from standard library

`tst rn` Modifies Z & N flags — `Z`, `N` reflect result

Assume the byte in `rn` represents a binary number and test whether it is zero or negative.
Used to modify flags without changing registers or memory.
The register `rn` remains unchanged by this operation, as do C and V.

`clr rn` `0` → `rn` — `C`, `V`, `Z`, `N` reflect result

Clear `rn`. This macro only takes one byte, to it is faster than `ldi rn, 0`

`shl rn`	(r*n* « 1) → r*n*)	C , V , Z , N reflect result - become 0

Sliced shift left r*n*
Shifts the bit-string in r*n* one place to the left without ensuring that the result is a multiple of two. The old value of C is shifted into bit 0, and the sign bit (bit 7) is shifted into C; V is 1 if bit 7 changes and 0 if it does not; N and Z are based on the modified r*n*.

`ldv const,rn`	**const* → r*n*	Flags unchanged

Gets a value from memory addr *const* to r*n*.

`stv rn,const`	r*n* → *const*	Flags unchanged

Puts r*n*contents to memory addr *const*

`ei`	0b10000000 → PS	C,V,Z,N become zero

Enable interrupts

`di`	0 → PS	C,V,Z,N become zero

Disable interrupts

A Cdm8e assembler grammar

A.1 Lexer rules

```
lexer grammar AsmLexer;

Asect : 'asect' ;
Break : 'break' ;
Continue : 'continue' ;
Do : 'do' ;
Else : 'else' ;
End : 'end' ;
Ext : 'ext' ;
Fi : 'fi' ;
Goto : 'goto' ;
If : 'if' ;
Is : 'is' ;
Macro : 'macro' ;
Restore : 'restore' ;
Rsect : 'rsect' ;
Save : 'save' ;
Stays : 'stays' ;
Then : 'then' ;
Tplate : 'tplate' ;
Until : 'until' ;
Wend : 'wend' ;
While : 'while' ;

Low : 'low' ;
High : 'high' ;

DOT : '.' ;
COMMA : ',' ;
PLUS : '+' ;
MINUS : '-' ;
COLON : ':' ;
ASTERISK : '*' ;
ANGLE_BRACKET : '>' ;
OPEN_PAREN : '(' ;
CLOSE_PAREN : ')' ;
LINE_MARK_MARKER: '-|';

REGISTER : 'r'[0-3] ;
WORD : [a-zA-Z_][a-zA-Z_0-9]* ;
DECIMAL_NUMBER : [0-9]+  ;
BINARY_NUMBER : '0b'[01]+ ;
HEX_NUMBER : '0x'[0-9a-fA-F]+ ;
STRING : '"'~["\\\n]*(('\\'.)~["\\\n]*)*'"' ;
CHAR : '\'' ('\\'. | ~[\\'\n]) '\'' ;

NEWLINE : '\r'? '\n' ;
COMMENT : '#'~[\n]* -> skip ;
WS : (' ' | '\t') -> skip ;
```

```
BASE64 : 'fp-' [a-zA-Z0-9/+=]+;


UNEXPECTED_TOKEN: [\u0000-\uFFFE];
```

A.2 Parser rules

```
parser grammar AsmParser;

options { tokenVocab=AsmLexer; }

@header {
from base64 import b64decode
}

@members {
    self.current_file = ''
    self.current_line = 0
    self.current_offset = 0
}

program : NEWLINE* line_mark+ section* End ;

section
    :  asect_header section_body # absoluteSection
    |  rsect_header section_body # relocatableSection
    | tplate_header section_body # templateSection
    ;

asect_header  :  Asect number NEWLINE+ ;
rsect_header  :  Rsect name   NEWLINE+ ;
tplate_header : Tplate name   NEWLINE+ ;

section_body : code_block ;
code_block
    :
    ( break_statement
    | continue_statement
    | line
    | conditional
    | while_loop
    | until_loop
    | save_restore_statement
    | goto_statement
    | line_mark
    )*
    ;

line_mark locals [
source_file = '',
```

```
source_line = 0
] : LINE_MARK_MARKER  line_number filepath  WORD? NEWLINE+ {
    self.current_line = int($line_number.text)
    self.current_file =  b64decode($filepath.text[3:]).decode()
    $source_file = self.current_file
    $source_line = self.current_line
    self.current_offset = $line_number.start.line - self.current_line + 1
};

line_number: DECIMAL_NUMBER;
filepath: BASE64;

break_statement : Break NEWLINE+ ;
continue_statement : Continue NEWLINE+ ;

line
    : label_declaration Ext? NEWLINE+                    # standaloneLabel
    | label_declaration? instruction arguments? NEWLINE+ # instructionLine
    ;

label_declaration: label (COLON | ANGLE_BRACKET) ;
arguments : argument (COMMA argument)* ;

conditional : If NEWLINE+ conditions code_block else_clause? Fi NEWLINE+ ;
conditions : connective_condition* condition NEWLINE+ (Then NEWLINE+)? ;
connective_condition : condition COMMA conjunction NEWLINE+ ;
condition : code_block Is branch_mnemonic ;
else_clause : Else NEWLINE+ code_block ;

branch_mnemonic : WORD ;
conjunction : WORD ;

while_loop : While NEWLINE+ while_condition Stays branch_mnemonic NEWLINE+
↪  code_block Wend NEWLINE+ ;
while_condition : code_block ;

until_loop : Do NEWLINE+ code_block Until branch_mnemonic NEWLINE+ ;

save_restore_statement : save_statement code_block restore_statement ;
save_statement : Save register NEWLINE+ ;
restore_statement : Restore register? NEWLINE+ ;

goto_statement : Goto branch_mnemonic COMMA goto_argument NEWLINE+ ;
goto_argument : addr_expr | byte_expr ;

argument
    : character
    | string
    | register
    | addr_expr
    | byte_expr
    ;
```

```
byte_expr : byte_specifier OPEN_PAREN addr_expr CLOSE_PAREN ;
addr_expr : first_term add_term* ;
first_term : (PLUS | MINUS)? term ;
add_term : (PLUS | MINUS) term ;
term : number | template_field | label ;
byte_specifier : Low | High ;

template_field : name DOT name ;
label : name ;
instruction : WORD ;
string : STRING ;
register : REGISTER ;
character : CHAR ;
number
    : DECIMAL_NUMBER
    | HEX_NUMBER
    | BINARY_NUMBER
    ;

name
    : Asect
    | Break
    | Continue
    | Do
    | Else
    | End
    | Ext
    | Fi
    | Goto
    | High
    | If
    | Is
    | Low
    | Macro
    | Restore
    | Rsect
    | Save
    | Stays
    | Then
    | Tplate
    | Until
    | Wend
    | While
    | WORD
    ;
```