CPU7 description

© KnivD@me.com 2015-16

document revision 4

Table of Contents

Introduction

CPU7 is a 14/16-bit processor with stack architecture. Its machine language is optimised for RPN (Reverse Polish Notation) languages, such as Forth. The internal registers are hidden from the end user thus increasing the convenience by programming in assembler similar to a higher level language, and still natively executed by the hardware. CPU7 also has built-in multi-threading capabilities at its very core level.

The internal structure of CPU7 is very simple and optimised for low transistor count ASIC realisation as well as a soft-core implementation into small FPGA devices. It also allows simple and effective software emulation in other processor platforms.

Due to its internal architecture CPU7 works most effectively with data which is wide in multiples of 14 bits – 14, 28, 42, and 56 bits, and it uses a 16-bit external data bus for communication with memory and peripheral interfaces.

The internal CPU7 registers are 56 bits wide and all arithmetic and logic operations are performed with the full length of the registers.

Data constants are represented in the memory as VLN (Variable Length Number), which take between one and four 16-bit words. VLN is further described in more details in this document.

There are a few generally valid rules in the work of CPU7:

- All words greater than a byte must be at aligned (even) address.
- CPU7 uses little endian format. Within a 16-bit word, the lower byte with bits 0..7 is stored at the first (even) address, and the higher byte with bits 8..15 is stored at the second (odd) address.
- CPU7 does not make difference between ROM, RAM and peripheral registers. Everything shares a single linear memory space.
- Each 16-bit word in the memory may consist of two 7-bit CPU instructions, two 8-bit bytes of raw data constants, or one 14-bit data constant.
- The two highest bits b.14 and b.15 are always reserved and used by CPU7 to determine the type of information stored in the word.
- If a new word differs by its type from the previous one, and the current address is an odd number, the remaining part of the word is padded with a NOP instruction (in case of instruction word) or with 0 (in case of data).
- Code execution always starts or restarts from memory address 0 and in the stack CPU7 leaves a word which carries the start-up condition code.

VLN is a way of storing numeric values with unknown length in the memory. CPU7 handles the operations at register level. VLN stores 14 bits of data in every 16-bit word, and the number of words only depends on the bit width of the expressed value. Since CPU7 registers are 56 bits wide, it can handle VLN up to four words long.

15 14	13 0
data type	*For VLN with less than 15 bits:* 14-bit data (bits 0..13)
data type	*For VLN with more than 14 bits:* 14-bit data (bits 14..27)
data type	*For VLN with more than 28 bits:* 14-bit data (bits 28..41)
data type	*For VLN with more than 42 bits:* 14-bit data (bits 42..55)

CPU7 allocates two system bits in every word in the program memory. These two bits define what type of data is stored in the remaining 14 bits in the word.

The following table shows the system bits and their meaning.

(bit) 15	*14*	
0	0	The word contains two 7-bit CPU7 instructions. The instruction stored in bits 0..6 is executed first, followed by the instruction stored in bits 7..13
0	1	Data word, not last in VLN
1	0	Data word, last in VLN. When data stored in VLN occupies only one word, it is also considered as last in VLN
1	1	CPU7 ignores the entire word and continues with the next one

When the execution of code starts from memory address 0, CPU7 leaves two words in the data stack. The top word carries a code which indicates the exact reason for the start or restart, and the one at stack position [1] indicates the memory address where the event has occurred. When the event is a cold start, the address is 0.

Code (hex)	*Description*
$100	Invalid instruction code
$101	Cold start
$102	Alignment error
$103	Invalid memory location
$104	Invalid data stack index
$105	Data stack overflow
$106	Data stack underflow
$107	Call stack overflow
$108	Call stack underflow
$109	Arithmetic error (eg. division by 0)
$10a	Error in a conditional structure ELSE or ENDIF without an opening IF UNTIL without an opening REPEAT
$10b	Double entering error
$10c	LEAVE without preceding ENTER

2. CPU7 instructions

2.1. Execution control and conditional structures

NOP

7-bit instruction code: *$7f*
Parameters and format: *NOP*

No operation.

Execution continues with the next instruction without any action taken.

DELAY

7-bit instruction code: *$1c*
Parameters and format: *x DELAY*

Delay of X nanoseconds for the current thread.

Execution is held for <u>at least</u> X nanoseconds while the execution of other parallel threads continues as normal. It can only be guaranteed that the delay is for no less than X nanoseconds as it depends on the execution times of the instructions in other parallel threads.

The actual delay counter resolution may vary from one CPU7 implementation to another, but in all cases the parameter of the DELAY instruction must be given in number of nanoseconds.

DO

7-bit instruction code: *$70*
Parameters and format: *DO*

Enables normal execution of the instructions following DO.

DO enables the execution <u>only</u> if its current execution nesting level is aligned with the preceding SKIP, otherwise it is treated as NOP.
Generally this means that DO must not be executed from within conditional structures or loops.

SKIP

7-bit instruction code: *$71*
Parameters and format: *x SKIP*

Skips <u>at least</u> X bytes and after that disables the execution until the next DO. All the following instructions are treated as NOP.

X must be an even number, and the instruction itself must be at even address.

Example:

```
      0 SKIP

        .........
        … code and data in this segment will be ignored …
        .........

      DO
```

IF

7-bit instruction code: $7c
Parameters and format: x IF

Marks the opening statement of a conditional branch structure. Takes a value from the stack and if the value is not 0 continues with the following instruction, otherwise moves the execution to the corresponding ELSE or ENDIF mark.

A IF structure may or may not have an alternative ELSE segment.

Example:

```
      x IF

        .........
        … code in this segment will be executed if x is not 0 …
        .........

      ELSE

        .........
        … code in this segment will be executed if x is 0 …
        .........

      ENDIF
```

ELSE

7-bit instruction code: $7e
Parameters and format: ELSE

Marks the alternative segment in a branch structure.

ENDIF

7-bit instruction code: $7d
Parameters and format: ENDIF

Marks the end of a conditional branch structure.

REPEAT

7-bit instruction code: $78
Parameters and format: REPEAT

Marks the opening statement of a loop structure.

The REPEAT instruction must be the only meaningful instruction within a 16-bit word. A padding NOP in front or after should be used.

REPIF

7-bit instruction code: $7b
Parameters and format: x REPIF

Conditional opening of a loop structure. Takes a value from the stack and executes REPEAT only if the value is not 0, otherwise ignores the code until a corresponding UNTIL or WHILE.

After the first time and if the closing statement has brought the execution back to REPIF, it performs as a normal REPEAT and nothing is taken from the stack.

The REPIF instruction must be the only meaningful instruction within a 16-bit word. A padding NOP in front or after should be used.

Example:

```
x REPIF

  .........
  … code in this segment will be executed for the first time I
         f x is not 0; the consequent iterations will depend on the
         value of y …
  .........

y UNTIL             `! if x was not 0 at the start, and y is not 0 now,
                    `! return to REPEAT, otherwise continue
```

REPIF is a functional equivalent of the following code:

```
x IF
  REPEAT

    .........

  y UNTIL/WHILE
ENDIF
```

UNTIL

7-bit instruction code: $79
Parameters and format: x UNTIL

Marks the closing statement of a loop structure. Takes a value from the stack and returns to the corresponding REPEAT if the value is 0, otherwise continues with the next instruction.

UNTIL can be used also to let a function conditionally cancel its return address or restart itself, that assuming the instruction is executed from within a function and not from within a loop structure.

Example:

```
REPEAT

  .........
  .........
  .........

x UNTIL             `! will return to REPEAT only if x is 0
```

WHILE

7-bit instruction code: $7a
Parameters and format: x WHILE

Marks the closing statement of a loop structure. Takes a value from the stack and returns to the corresponding REPEAT if the value is not 0, otherwise continues with the next instruction.

WHILE can be used also to let a function conditionally cancel its return address or restart itself, that assuming the instruction is executed from within a function and not from within a loop structure.

Example:

```
REPEAT

  .........
  .........
  .........

x WHILE             `! will return to REPEAT only if x is not 0
```

BREAK

7-bit instruction code: $03
Parameters and format: x BREAK

Exits the current loop structure if the parameter is not 0.

AGAIN

7-bit instruction code: $04
Parameters and format: x AGAIN

Unconditionally returns to the corresponding REPEAT or REPIF statement, if the parameter is not 0.

Example:

```
      REPEAT

        .........
        … code in this segment will be executed …
        .........

        1 AGAIN

        .........
        … code in this segment will not be executed because of the AGAIN instruction …
        .........

      x WHILE             `! execution will never reach this statement
```

CALL

7-bit instruction code: $05
Parameters and format: x CALL

Stores the current return address, takes a value from the stack and uses is to calculate a new address where the execution will be taken.
Since the value is subtracted from the current address, the new address is always smaller.

If the CALL instruction is at an even address, the following instruction which will be an odd address, must be NOP, because the execution will return at the following even address after the CALL.

The call address must be 16-bit word aligned (even number).

Example:

```
[$1000]      $400 CALL    `! will store the current return address $1002 in the
                          `! call stack, and will take the execution to address $c02
```

ACALL

7-bit instruction code: $06
Parameters and format: x ACALL

Stores the current execution address, takes a value from the stack and takes the execution to that absolute address.

If the CALL instruction is at an even address, the following instruction which will be an odd address, must be NOP, because the execution will return at the following even address after the CALL.

The call address must be 16-bit word aligned (even number).

NOTE: This instruction must be avoided at any possible cost. Its only reason for existence is to allow support of pre-built in ROM libraries, which have functions at always known address.

Example:

```
[$1000]      $400 ACALL   `! will store the current return address $1002 in the
                          `! call stack, and will take the execution to address $400
```

NTCALL

7-bit instruction code: $0d
Parameters and format: x NTCALL

Creates a new thread, which will start from the address which is relative back to the current one by a number of bytes taken from the stack.
Since it creates a new thread, no return address is stored in the call stack.

The call address must be 16-bit word aligned (even number).

Example:

```
[$1000]      $400 NTCALL  `! will create a new parallel thread which starts from
                          `! address $c02. The current thread continues with the
                          `! execution of the next instruction
```

NTACALL

7-bit instruction code: $0e
Parameters and format: x NTACALL

Creates a new thread, which will start from absolute address which is taken from the stack.
Since it creates a new thread, no return address is stored in the call stack.

The call address must be 16-bit word aligned (even number).

NOTE: This instruction must be avoided at any possible cost. Its only reason for existence is to allow support of pre-built in ROM libraries, which have functions at always known address.

Example:

```
[$1000]      $400 NTACALL `! will create a new parallel thread which starts from
                          `! address $400. The current thread continues with the
                          `! execution of the next instruction
```

RETURN *(also possible ;)*

7-bit instruction code: $07
Parameters and format: RETURN   (also ;)

End of function. Restores the execution address from the call stack.

MAXTHDS

7-bit instruction code: $08
Parameters and format: MAXTHDS

Returns in the stack the number of maximum supported parallel threads.

THREADS

7-bit instruction code: *$09*
Parameters and format: *THREADS*

Returns in the stack the number of currently executing parallel threads.

ENDALL

7-bit instruction code: *$0b*
Parameters and format: *ENDALL*

End all parallel threads except for thread 0 and resets the CPU7 registers associated with them.

END

7-bit instruction code: *$0c*
Parameters and format: *END*

End the current thread and resets the CPU7 registers associated with it. For thread 0 performs a normal RETURN operation.

SETPR

7-bit instruction code: *$01*
Parameters and format: *x SETPR*

Every new thread normally runs with priority 1, which means it gets one instruction timeslot (two instructions). When needed a thread can increase or decrease its own priority thus increasing or decreasing the size of its own timeslot. This will effectively affect the overall execution speed of the thread.

The minimum possible value of X for a working thread is 1, which corresponds to a single instruction word with two CPU7 instructions. A value 0 will terminate the thread just like the instruction END does.

Very large values of X will result in “choppy” execution of all other threads and under normal conditions it is recommended that the value of X is not greater than 10 times the maximum number of available threads in a system.

ENTER

7-bit instruction code: $18
Parameters and format: ENTER

Takes a snapshot of the current data stack pointer. This is used to enter a zone where the data stack can be used for local variables.

LEAVE

7-bit instruction code: $19
Parameters and format: LEAVE

Restores the data stack pointer from a previously taken snapshot.

2.2. Operations with the hardware data stack

EMPTY

7-bit instruction code: $11
Parameters and format: EMPTY

Empties the data stack.

DEPTH

7-bit instruction code: $12
Parameters and format: DEPTH

Returns the number of elements currently in the data stack. The returned value is also taken in count.

DROP

7-bit instruction code: $13
Parameters and format: DROP

Removes the top stack element.

DUP

7-bit instruction code: $14
Parameters and format: DUP

Duplicates the top stack element.

x SWAP

7-bit instruction code: $15
Parameters and format: x SWAP

Swaps the top stack element (after retrieving the parameter X) with the one at Xth depth position in the stack. Since the top element is referred to as in position 0, a "*0 swap*" instruction will do nothing, "*1 swap*" will swap the top two elements, etc.

ROT

7-bit instruction code: $16
Parameters and format: ROT

Rotates positions of the three top stack elements.

Example:

```
10 20 30     `! the stack (from top downwards) is 30, 20, 10
ROT          `! the stack (from top downwards) is 10, 30, 20
```

OVER

7-bit instruction code: $17
Parameters and format: x OVER

Copies the xth stack element (after taking the parameter x) on top. Counting starts from 0, hence 0 OVER is a functional equivalent of DUP.

Example:

```
10 20 30     `! the stack (from top downwards) is 30, 20, 10
1 OVER       `! the stack (from top downwards) is 20, 30, 20, 10
```

!

7-bit instruction code: $1a
Parameters and format: x !

Reads variable register x (where x is a valid number for register Vx) into the data stack.

=!

7-bit instruction code: $1b
Parameters and format: v x =!

Writes the value v into a variable register x (where x is a valid number for register Vx).

2.3. Logic and bitwise operations

COM

7-bit instruction code: *$20*
Parameters and format: *x COM*

Returns the 56-bit complementary value of the parameter. The complementary value is calculated by bitwise negating and adding 1 to the result.

NOT

7-bit instruction code: *$21*
Parameters and format: *x NOT*

Returns the 56-bit bitwise negated value of the parameter.

AND

7-bit instruction code: *$22*
Parameters and format: *x y AND*

56-bit bitwise AND operation of the two top stack elements.

OR

7-bit instruction code: *$23*
Parameters and format: *x y OR*

56-bit bitwise OR operation of the two top stack elements.

XOR

7-bit instruction code: *$24*
Parameters and format: *x y XOR*

56-bit bitwise “Exclusive OR” operation of the two top stack elements.

SHL

7-bit instruction code: $26
Parameters and format: x y SHL

56-bit bitwise shift left of X by a given number of bits Y. On each iteration the most significant bit 55 is lost, and the least significant bit 0 is loaded with 0.

SHR

7-bit instruction code: $27
Parameters and format: x y SHR

56-bit bitwise shift right of X by a given number of bits Y. On each iteration the least significant bit 0 is lost, and the most significant bit 55 is loaded with 0.

2.4. Arithmetic operations

<

7-bit instruction code: $28
Parameters and format: x y <

Compare the two top stack values and returns 1 if the one at position 1 is smaller than the one at position 0, otherwise returns 0.

Example:

```
      10 20 <      `! will return 1, because 10 is smaller than 20
      55 50 <      `! will return 0, because 55 is not smaller than 50
```

<=

7-bit instruction code: $29
Parameters and format: x y <=

Compare the two top stack values and returns 1 if the one at position 1 is smaller or equal to the one at position 0, otherwise returns 0.

Example:

```
      10 20 <=     `! will return 1, because 10 is smaller or equal than 20
```

==

7-bit instruction code: $2a
Parameters and format: x y ==

Compare the two top stack values and returns 1 if they are equal, otherwise returns 0.

<>

7-bit instruction code: $2b
Parameters and format: x y <>

Compare the two top stack values and returns 1 if they are not equal, otherwise returns 0.

>=

7-bit instruction code: $2c
Parameters and format: x y >=

Compare the two top stack values and returns 1 if the one at position 1 is greater or equal to the one at position 0, otherwise returns 0.

Example:

```
30 20 >=     `! will return 0, because 30 is greater or equal than 20
```

>

7-bit instruction code: $2d
Parameters and format: x y >

Compare the two top stack values and returns 1 if the one at position 1 is greater or equal to the one at position 0, otherwise returns 0.

Example:

```
30 20 >      `! will return 1, because 30 is greater than 20
20 50 >      `! will return 0, because 20 is not greater than 50
```

+

7-bit instruction code: $40
Parameters and format: x y +

Returns the arithmetic sum of the two top stack elements.

-

7-bit instruction code: $42

Parameters and format: x y -

Subtracts the top element from the one at position 1 and returns the result.

*

7-bit instruction code: $44
*Parameters and format: x y **

Multiplies the two top stack elements and returns the result.

/

7-bit instruction code: $46
Parameters and format: x y /

Stack element [1] is divided by stack element [0] (the top) and the result is returned into the stack.

//

7-bit instruction code: $47
Parameters and format: x y //

Modulo operation. Stack element [1] is divided by stack element [0] (the top) and the reminder of the integer division is returned into the stack.

++

7-bit instruction code: $48
Parameters and format: x ++

The top stack element is increased by 1.

--

7-bit instruction code: $49
Parameters and format: x --

The top stack element is decreased by 1.

RANDOM

7-bit instruction code: $4b
Parameters and format: RANDOM

A 56-bit random value is pushed into the stack.

2.5. Operations with the memory

FILL

7-bit instruction code: $50
Parameters and format: a c v FILL

Fill a memory block with start address A and length C with the value V.

DIFF

7-bit instruction code: $52
Parameters and format: x y c DIFF

Compares two memory blocks with length C, one starting from address X, and other from address Y.
Returns 0 if they are not different, otherwise returns the number of remaining to test characters where the first difference has been found.

=

7-bit instruction code: $53
Parameters and format: a n c =

Copies a memory block with size C from address A to address N. Overlapping is allowed in both directions.

LEN$

7-bit instruction code: $58
Parameters and format: a LEN$

Returns the length of a zero-terminated string starting from address A. The returned length excludes the terminating $00 character.

SCAN$

7-bit instruction code: $59
Parameters and format: x y SCAN$

Search for the first occurrence of the zero-terminated string Y in X.
Returns the memory address of the beginning of Y in X, or 0 if not found.

NOTE: the result of this function makes it impossible to distinguish between Y being found at absolute memory address 0, and not being found at all. The user software must be aware of this fact, although it is an incredibly unlikely situation.

DIFF$

7-bit instruction code: *$5a*
Parameters and format: *x y DIFF$*

Compares two zero-terminated memory blocks, one starting from address X, and other from address Y.
Returns 0 if they are not different, otherwise returns the number of tested and equal characters.

=$

7-bit instruction code: *$5b*
Parameters and format: *a n $=*

Copies a zero-terminated block from address A to address N. Allows overlapping in both directions.

RDVLN

7-bit instruction code: *$60*
Parameters and format: *a RDVLN*

Reads and returns to stack[0] a VLN value, read from address A. Also returns in stack[1] the address following the value.

RD32

7-bit instruction code: *$64*
Parameters and format: *a RD32*

Reads and returns to stack a “raw” 32-bit value, read from address A.

RD16

7-bit instruction code: *$65*
Parameters and format: *a RD16*

Reads and returns to stack a “raw” 16-bit value, read from address A.

RD8

7-bit instruction code: $66
Parameters and format: a RD8

Reads and returns to stack a “raw” 8-bit value, read from address A.

WRVLN

7-bit instruction code: $68
Parameters and format: x a WRVLN

Writes the top stack element as a VLN value at memory address A. Then returns in the stack the address following the written VLN.

WR32

7-bit instruction code: $6c
Parameters and format: x a WR32

Writes the top stack element as a “raw” 32-bit value at memory address A. If X is wider than 32 bits, only the lowest 32 bits are written and the rest is ignored.

WR16

7-bit instruction code: $6d
Parameters and format: x a WR16

Writes the top stack element as a “raw” 16-bit value at memory address A. If X is wider than 16 bits, only the lowest 16 bits are written and the rest is ignored.

WR8

7-bit instruction code: $6e
Parameters and format: x a WR8

Writes the top stack element as a “raw” 8-bit value at memory address A. If X is wider than 8 bits, only the lowest 8 bits are written and the rest is ignored.

2.6. System functions

SYSFN

7-bit instruction code: *$17*
Parameters and format: *x SYSFN*

Calls a 'system function' with code X.

Unknown function codes are ignored.

System function can be any operation which is performed outside of the CPU7 model. As such it could be for example a complex mathematical function performed by a dedicated hardware, graphics, interface, initialisations, etc.

System functions are not executed by CPU7 (with the exception of codes smaller than $80), but are done in hardware, microcode or the host processor in case of emulation.

Function codes $00..$7f are reserved and executed by CPU7 in the way each code executing the exact CPU7 instruction represented with it.

Codes $80 and greater are unspecified and strictly dependent on the exact system implementation. These codes are not generally transferable amongst different CPU7 realisations.

What the system functions take or leave in data stack, except codes lower than $80, which are the standard CPU7 instructions with pre-defined behaviour, is also undefined and specific for the implementation.

Another important point for consideration is that the system functions are executed as integral instructions. As such they occupy a single execution timeslot in the CPU7's schedule. A system function which takes very long time for completion will hold all other threads while executing.

3. CPU7 register map

```
                                                       +--------+
                          7-bit process index register |  PXR   |
     +-------------------------------------------------+--------+
     |   56-bit DLYC (free-running incremental delay counter)   |
     +----------------------------------------------------------+

                               . . . . . . . . . . . . . . .. . .
                             +----------------------------+ . . .
        program code pointer |         28-bit PCP        0| . . .
+----------------------------+----------------------------+ . . .
|         56-bit CAR (conditional action register)      **| . . .
+----------------------------+----------------------------+ . . .
|                    56-bit register R0 (A)               | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register R1 (B)               | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register R2 (C)               | . . .
+------------------------------------------+--------------+ . . .
                                           | 14-bit R3 (D)| . . .
                                           +--------------+ . . .
                                           | 14-bit R4 (E)| . . .
                                           +--------------+ . . .
                 instruction code register | 14-bit ICR   | . . .
                                           +--------------+ . . .
                        call stack pointer | 14-b CSP   00| . . .
                                           +--------------+ . . .
                        data stack pointer | 14-b DSP  000| . . .
                                           +--------------+ . . .
               data stack pointer snapshot | 14-b SSP  000| . . .
                                           +--------------+ . . .
                  data stack depth counter | 14-bit DDC   | . . .
                                           +--------------+ . . .
         data stack depth counter snapshot | 14-bit SDC   | . . .
                                           +--------------+ . . .
                 process priority register | 14-bit PPR   | . . .
+------------------------------------------+--------------+ . . .
|             56-bit DCR (delay compare register)         | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register V0                   | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register V1                   | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register V2                   | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register V3                   | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register V4                   | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register V5                   | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register V6                   | . . .
+---------------------------------------------------------+ . . .
|                    56-bit register V7                   | . .
+---------------------------------------------------------+ .
```

4. Torth translator

CPU7's machine language on its own is very close to a high level RPN language similar to Forth, so the Torth translator does very little more than simply converting the instruction mnemonics into data words, and converting text labels into memory addresses. In fact Torth is the native programming language for CPU7.

There are a few additional features which Torth offers to the CPU7 machine code.

:*label*
:*label @loc*

Mark the current program memory address and assign it to the name *label*. Later in the program every reference to *label* will insert the marked address into the data stack.

If the parameter *@loc* is present immediately following the definition, *label* will be assigned to the specific location in the memory map at absolute address *loc*.

.*label*

Insert the memory address associated with *label* into the data stack during execution.

_*label*	call (translated as .*label* CALL)
_!*label*	multi-threaded call (translated as .*label* NTCALL)
&*label*	absolute call (translated as .*label* ACALL)
&!*label*	absolute multi-threaded call (translated as .*label* NTACALL)

` ....... commentaries (until the next ` character) .... `
`! ...... commentaries until the end of the current source line only

"*filename*" **INCLUDE**

Include Torth source from an external file. The input file is a text file and it gets inserted into the exact location where the INCLUDE command is in the current source.

"*filename*" **LIBRARY**

Include CPU7 binary from an external file. The included file is a CPU7 binary file which is directly inserted at the current memory location.