vCPU Documentation v1.1

by Dennis Kuschel, 2016-09-17

Contents

Introduction

vCPU is a 16-bit extension to MyCPU. The simulated CPU has access to all the memory of MyCPU in a flat memory model (without bank-switching). This enables the development of large programs that need access to lots of memory. Because vCPU has also access to the processor interrupts it is also possible to write a complete Operating System for MyCPU.

vCPU is realized as an interpreter that interprets the 16-bit instructions. The interpreter is highly optimized for speed to get best performance. When your application is time-critical you should have a look to the 2nd version of vCPU, namely vCPU2 (see page 12).

Example Program

Programs for vCPU can be assembled with the myca cross assembler. Below is an example listing for a "hello world" program. Assuming the file is named "`hello.asm`", you can translate it into a binary with the command "`myca hello.asm -o hello -l`". The option `-l` is optionally, it advices myca to generate a listfile with the name `hello.lst`.
To run a vCPU program simply type its name on the MyCPU command prompt. To run the program in the debugger, type "`vcpud programname`" at the prompt.

Program Listing

```
;==============================================================================
.target vcpu                ; tell myca to assemble for vCPU
.mode ascii                 ; switch to ASCII mode, since vCPU is ASCII compatible
vcpu_stack_size set 10*1024 ; set size of stack to 10 kbyte
vcpu_heap_size  set 64*1024 ; set size of heap to 64 kbyte
codeseg segment code        ; generate a new code segment with the name codeseg
org 0                       ; bind the segment to address 0x000000
;==============================================================================

; Place all data into the dataseg segment.
; myca will place the data automatically behind the program code.
dataseg segment code

text_hello  DB "Hello World!\n", 0


; The program code is filled into the codeseg segment:
codeseg segment code

main:   ;main program, starts at address 0x000010
        ldp   p0,#text_hello
        sout  p0
        ret
```

The Program Input Registers

When the vCPU interpreter starts executing a program, it sets up the vCPU environment (memory, registers and flags) and jumps to program address 0x00000000. Some registers are used to pass some additional information to the vCPU program:

r0 - r6	set to 0x0000
r7	vCPU interpreter version
p0 (r8,r9)	pointer to the command line string
p1 (r10,r11)	pointer to start of heap memory (first byte)
p2 (r12,r13)	pointer to end of heap memory (last byte)
sp (r14,r15)	set to top of the stack memory
flags (r15)	C=0, Z=0

The Magic Program Header

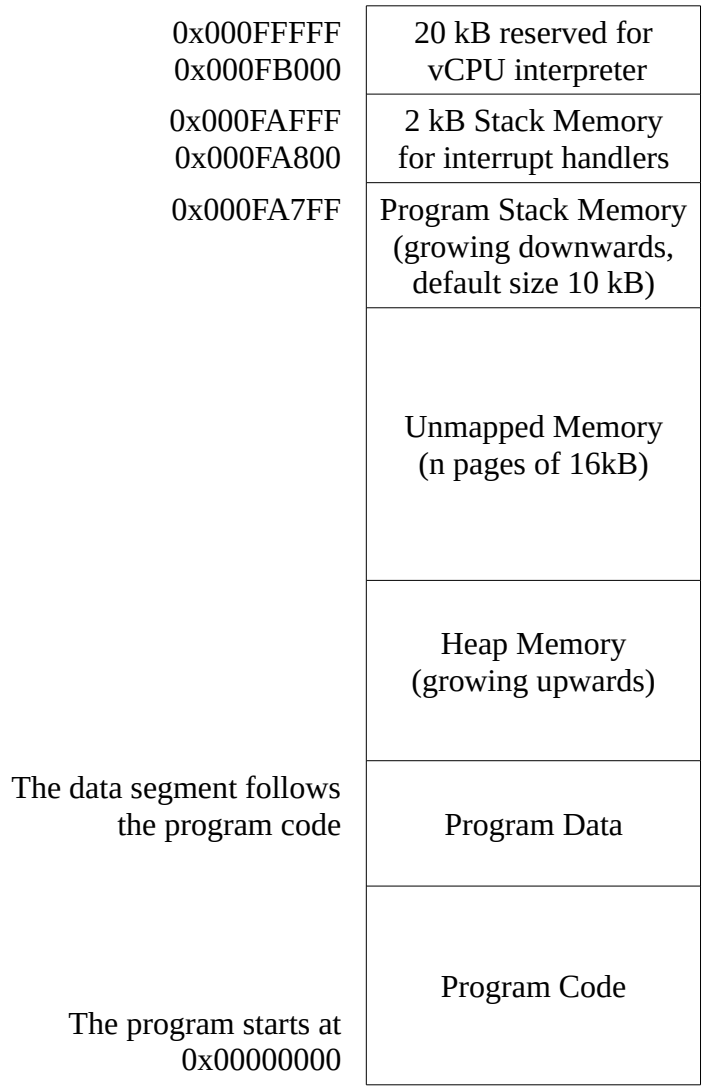
A vCPU program must be prefixed with a 16-byte header. The header tells the MyCPU Operating System the type of program and the name of the loader that is used to load the program into memory. The text-segment of the program itself starts directly behind this header. The vCPU interpreter starts program execution at address 0, and the call-instruction at this address will jump to the first instruction in the text-segment at address 0x00000010 directly behind this header. The cross-assembler “myca” inserts the magic program header automatically for programs that start at address 0.

Offset	**Bytes**	**Meaning**
0 - 3	0x11, 0x00, 0x10, 0x00	OP-Code for "CALL 0x00000010"
4	0x06	OP-Code for "HALT"
5	0x00 by default	bit 0-5 : Number of pre-allocated heap-pages bit 6-7 : Number of additional stack-pages Notes: - a page has a size of 16kB - max. stack size is 10kB + 3 * 16kB = 58kB
6 - 13	0x4C, 0x44, 0x52, 0x3A, 0x56, 0x43, 0x50, 0x55	Name of the loader: "LDR:VCPU"
14	0x00	Reserved character for the loader name
15	0x00	Terminator for the loader name

Memory Map

Overview

The figure below shows the memory map. The whole address range is from 0 to 0x000FFFFF, this gives a total of 1 MB addressable memory. The memory is divided into pages of 16 kB, and not all memory pages are available to vCPU. This means that the overall usable memory is a bit less than 1 MB: The operating system itself uses 6 memory pages, thus only 928 kB remain for a vCPU program. But vCPU can be switched into an "exclusive" mode which allows vCPU to access all the MyCPU resources. In this mode all memory pages are available for vCPU, and the vCPU program is the only program that runs on MyCPU.



Stack Memory

vCPU knows two kinds of data stack memory: The interrupt stack has a fixed size of 2 kB and is used by interrupt service routines. The application program stack has a default size of 10 kB. It can be increased by a myca assembler setting. The two stacks share a common 16 kb memory page. The upper 4 kB of the page are used by the vCPU interpreter for data-exchange with the kernel ROM. In "Exclusive Access Mode" (page 10) this memory is also used for the interrupt stack. The stack contains only data. The return-addresses of subroutine calls are stored on the MyCPU processor stack that is outside of vCPU's visible memory range. vCPU has a special instruction named “STK” (see page 70) that enables the user to access the call-stack.

Heap Memory

No heap is allocated by default. But since the granularity of the memory pages is 16kb, some free memory remains behind the program data area that can be used for the heap. Thus, if no heap is defined for a program, a program will often have access to a bit heap memory. A program can use the system-call function 0x80 to allocate more memory pages for the heap.

Register Set

The simulated CPU has 16 registers that are 16-bit wide. The registers r0 to r13 are multi purpose registers. The registers r14 and r15 are special: register 14 and the low part of register 15 contain the 24 bit stack-pointer. The upper 8 bits of register 15 contain the processor flags: Bit 14 of register 15 is the zero flag, bit 15 is the carry flag. The bits 8 to 13 are reserved for future use and are always zero.
Two consecutive registers can be combined to a single 32-bit register. The registers r0 to r7 are multi purpose data registers (either 8 16-bit registers or 4 32-bit registers, or any combination of that) The 32-bit pseudo registers can be accessed by the register names d0 - d3. The registers r8 to r13 are defined to be pointer registers, which can be accessed by their names p0 - p2. To select the upper or lower 16-bit part of the register, the myca assembler knows the notation d0.l and d0.h (p0.l and p0.h respectively). For better readable source code the stack pointer register r14/r15 can be accessed by its alias "sp". The register r15 is shared and contains also the processor flags. But if r14/r15 is used to access the stack memory, the upper 8 bits (=flags) are masked out.

16-bit Register Name	Alternative Name	Alternative Usage
`r0`	`d0, d0.l`	32-bit data register 0 (lower 16-bit)
`r1`	`d0.h`	32-bit data register 0 (upper 16-bit)
`r2`	`d1, d1.l`	32-bit data register 1 (lower 16-bit)
`r3`	`d1.h`	32-bit data register 1 (upper 16-bit)
`r4`	`d2, d2.l`	32-bit data register 2 (lower 16-bit)
`r5`	`d2.h`	32-bit data register 2 (upper 16-bit)
`r6`	`d3, d3.l`	32-bit data register 3 (lower 16-bit)
`r7`	`d3.h`	32-bit data register 3 (upper 16-bit)
`r8`	`p0, p0.l`	32-bit pointer or address register 0 (lower 16-bit)
`r9`	`p0.h`	32-bit pointer or address register 0 (upper 16-bit)
`r10`	`p1, p1.l`	32-bit pointer or address register 1 (lower 16-bit)
`r11`	`p1.h`	32-bit pointer or address register 1 (upper 16-bit)
`r12`	`p2, p2.l`	32-bit pointer or address register 2 (lower 16-bit)
`r13`	`p2.h`	32-bit pointer or address register 2 (upper 16-bit)
`r14`	`sp, sp.l`	24-bit stack pointer, lower 16-bit
`r15`	`sp.h`	bit 0-7: 24-bit stack pointer, upper 8 bit Processor Flags: bit 14: Zero-Flag bit 15: Carry-Flag

Exclusive Access Mode

vCPU supports two modes of operation: The usual program mode and the exclusive access mode. The vCPU is fired up in usual program mode but can be switched to exclusive access mode with the EXCL-instruction.

Usual Program Mode:
The application program has only access to currently mapped memory. If the program requires more memory, it must call a system function to increase the heap size. Furthermore it has only limited access to the underlying hardware. This is because the MyCPU operating system runs in parallel and provides its own drivers for the hardware. The vCPU application program can call MyCPU kernel functions for File-I/O or VGA Screen Output. Theoretically multiple vCPU programs can run in parallel.

Exclusive Access Mode:
All available memory is mapped and thus available to the application program. The program has full access to the MyCPU hardware. Because of this the MyCPU operating system kernel does no more run, and no hardware driver services are available. The vCPU program has to provide its own driver functions to access serial I/O, the VGA graphics unit and the IDE controller.

Jump-Tables

Sometimes it is necessary to implement so-called “jump-tables”. The usual application is the communication or linkage between two programs. For example if your vCPU program wants to load another program (maybe a library) into memory, it is required to know the entry points of the exported functions. Jump-Tables are a bit tricky on vCPU2, because the cross-assembler will not always generate Jump-OP-Codes with the same size. Jumps on vCPU are 4 bytes long, and jumps on vCPU2 are 3 to 8 bytes long. So it is required to force all jumps to a constant size of 8 bytes. This is done with the align_code - command. Note that there is another restriction on vCPU2: Since every code-segment of 32kb has its own runtime-library-block at the beginning, there are some restrictions where the jump-table can be placed. Preferably the jump-table should be placed at an offset of 0x1000 to the start address of the program. Here is an example:

```
;==================================================================================
.target vcpu2               ; tell myca to assemble for vCPU2
.mode ascii                 ; switch to ASCII mode, since vCPU is ASCII compatible
vcpu_stack_size set 10*1024 ; set size of stack to 10 kbyte
vcpu_heap_size  set  0*1024 ; set size of heap to 0
codeseg segment code        ; generate a new code segment with the name codeseg
org 0x40000                 ; bind the segment to address 0x00040000
;==================================================================================

codeseg segment code

        jump  main                  ;jump behind the table

        align_code 0x1000           ;force jump-table to start at address 0x00041000

        jump  exported_function_1   ;assembly-address is 0x00041000
        align_code 8
        jump  exported_function_2   ;assembly-address is 0x00041008
        align_code 8
        jump  exported_function_3   ;assembly-address is 0x00041010
        align_code 8
          :
          :
        jump  exported_function_n-1
        align_code 8
        jump  exported_function_n


main:   ;main program
        ret
```

Note:
Of course you could also use jumps over pointers (e.g. if your library exports a data structure that lists all exported functions), but this is really no good idea on vCPU2. This is because vCPU2 must translate the memory address for every pointer, and that makes function calls over pointers incredibly slow.

Avoid function pointers where possible!

vCPU2

vCPU version 2 (short: vCPU2) is source-code-compatible to vCPU, but it is completely different implemented. vCPU is implemented as interpreter, whereas vCPU2 is not really a virtual CPU but an extension to the cross-assembler myca. The cross-assembler can translate vCPU source code directly to a native MyCPU op-code stream. Because of this vCPU2 programs are around three to four times faster than vCPU programs, but they are also around four times larger than vCPU programs.

A vCPU program can easily be translated to vCPU2 by two ways:
- replace the "`.target vcpu`" command in the source file by "`.target vcpu2`"
- add the option "`-t vcpu2`" to the invocation of myca

Comparison of vCPU and vCPU2

	vCPU	**vCPU2**
Architecture	CPU Emulator / Interpreter	native MyCPU code
Debugger available	yes	no
Code size per OP-Code	2 or 4 bytes	undefined, up to 24 bytes
Speed	approx. 27,000 OPs/sec	4 x faster than vCPU
Maximum available RAM	1008 kB	1008 kB
Support for Hardware Interrupts	yes	yes
Suited to implement an OS	yes	yes

Command-line Options for vCPU2

The cross-assembler knows some additional options for vCPU2 programs:

-lnv	Do not list vCPU instructions in the listfile
-O0	Do not optimize the code
-O1	Optimize code
-O2	Optimize code, remove unneeded instructions (this is the default)
-Os	Optimize the code for size (it will get slower)
-Ofast	Optimize the code for speed (it will get larger)

vCPU2 Fast Pointers

Usually the register pairs r8/r9, r10/r11 and r12/r13 are used as pointer registers p0, p1 and p2 in vCPU assembler code. Now vCPU2 supports a faster version of these pointers. This is done by doing the necessary address translation only once, and that is when the pointer register gets loaded or modified. myca knows a special target command to enable the fast pointers on demand:

`.targetop fastptr-p0 on`	enables fast pointer code for p0
`.targetop fastptr-p0 off`	disables fast pointer code for p0

To temporary enable fast pointers you can write this:

`.targetop fastptr-p0 push,on`	saves last state and enables fast pointer code for p0
`.targetop fastptr-p0 pop`	restores last fast pointer state

This lines temporary disable fast pointers:

`.targetop fastptr-p0 push,off`	saves last state and disables fast pointer code for p0
`.targetop fastptr-p0 pop`	restores last fast pointer state

To enable fast pointer support for p1 and p2 you write:

```
.targetop fastptr-p1 on
.targetop fastptr-p2 on
```

Example:

```
.targetop fastptr-p0 push,on
ldd  p0,#structure
ldw  r1,(p0)
ldw  r2,(p0+4)
ldw  r3,(p0+12)
.targetop fastptr-p0 pop
```

Here is an example that will <u>not</u> work because the necessary address translation is not done (this is because pointer register p2 gets loaded before p2 is declared as fast-pointer):

```
ldd  p2,#structure
.targetop fastptr-p2 push,on
ldw  r5,(p2+200)
.targetop fastptr-p2 pop
```

Note:
You should use fast pointers only to access data structures several times with the same pointer like shown in the working example above. If only one element gets accessed after the pointer was loaded, you should better use standard pointers to save CPU time. For example, this simple copy loop will be faster with fast pointers disabled:

```
for (i=0;i<10;i++) { *p0++ = *p1++; }
```

But this code will be faster with p1 set as fast pointer because p1 is accessed twice:

```
while (*p1 != 0) { *p0++ = *p1++; }
```

Instruction Set

The vCPU supports up to 128 instructions. Instructions have a size of 16 or 32-bit. vCPU is a little endian machine, so the least significant bit and the least significant byte is stored first.

The following pages list the instruction set supported by vCPU and vCPU2. The binary representation of the encoding is not valid for vCPU2.

Note:
Bits that are marked with a dash (-) are don't cares, they can be 0 or 1. The byte that contains the OP-Code is the first byte in memory, followed by the register-name-byte and optionally by the two bytes that contain an absolute address for the 4-byte OP-Codes.

ADC r,r (add with carry)

OP-Code: 0x53
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	0	0	1	1	y	y	y	y	x	x	x	x

Adds the contents of register x, register y and the carry flag. If the operation overflows the carry flag gets set, otherwise the carry flag gets cleared. The result of the operation is stored in register x.

x = register name r0-r15 of operand 1 and destination register
y = register name r0-r15 of operand 2

Affected flags: carry flag

Example: ADC r4,r7

ADC r,# (add with carry)

OP-Code: 0x73
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	0	0	1	1	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Adds the contents of register r, an immediate value and the carry flag. If the operation overflows the carry flag gets set, otherwise the carry flag gets cleared. The result of the operation is stored in register r.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: carry flag

Example: ADC r1,#0x0150

ADCD r,r (add with carry, 32-bit)

OP-Code: 0x5B
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	1	0	1	1	y	y	y	y	x	x	x	x

Adds the contents of register `x` and `x+1`, register `y` and `y+1` and the carry flag. If the operation overflows the carry flag gets set, otherwise the carry flag gets cleared. The result of the operation is stored in register `x` and `x+1`.

x = register name r0-r15 of operand 1 and destination register, e.g. r=4 for d2
y = register name r0-r15 of operand 2, e.g. r=6 for d3

Affected flags: carry flag

Example: ADCD d2,d3

ADD r,# (add)

OP-Code: 0x78
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	1	0	0	0	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Adds an immediate unsigned value to the contents of register `r`. If the operation overflows the carry flag gets set, otherwise the carry flag gets cleared. The result of the operation is stored back in register `r`.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: carry flag

Example: ADD r1,#0x31B4

ADDD r,r (add, 32-bit)

OP-Code: 0x4D
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	1	1	0	1	y	y	y	y	x	x	x	x

Adds the contents of register `y` and `y+1` to the register `x` and `x+1`. The result of the operation is stored in register `x` and `x+1`.

x = register name r0-r15 of operand 1 and destination register, e.g. r=4 for d2
y = register name r0-r15 of operand 2, e.g. r=6 for d3

Affected flags: none

Example: ADDD d2,d3

ADDD r,# (add, 32-bit)

OP-Code: 0x6E
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	0	1	1	1	0	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Adds an immediate unsigned value to the contents of register `r` and `r+1`. The result of the operation is stored back in register `r` and `r+1`.

r = register name r0-r15
h = bits 15-8 of the immediate value
l = bits 7-0 of the immediate value

Affected flags: none

Example: ADDD d2,#0x1582

AND r,r (logical AND)

OP-Code: 0x50
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	0	0	0	0	y	y	y	y	x	x	x	x

Does a logical AND of the contents of register `x` and register `y`. The result of the operation is stored in register `x`.

x = register name r0-r15 of operand 1 and destination register
y = register name r0-r15 of operand 2

Affected flags: none

Example: AND r3,r5

AND r,# (logical AND)

OP-Code: 0x70
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	0	0	0	0	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Does a logical AND of the contents of register `r` and an immediate value. The result is stored back in register `r`.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: none

Example: AND r4,#0xF630

ANDD r,r (logical AND, 32-bit)

OP-Code: 0x58
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	1	0	0	0	y	y	y	y	x	x	x	x

Does a logical AND of the contents of register `x,x+1` and register `y,y+1`. The result of the operation is stored in register `x` and `x+1`.

x = register name r0-r15 of operand 1 and destination register, e.g. r=4 for d2
y = register name r0-r15 of operand 2, e.g. r=6 for d3

Affected flags: none

Example: ANDD d2,d3

ASR r (arithmetic shift right)

OP-Code: 0x2A
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	1	0	1	0	-	-	-	-	r	r	r	r

Shift the content of register `r` right, with sign extension. The overflowing LSB is shifted into the carry flag.

r = register name r0-r15

Affected flags: carry flag

Example: ASR r6

ASRD r (arithmetic shift right, 32-bit)

OP-Code: 0x2B
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	1	0	1	1	-	-	-	-	r	r	r	r

Shift the content of register `r` and `r+1` right, with sign extension. The overflowing LSB is shifted into the carry flag.

r = register name r0-r15, e.g. r=4 for d2

Affected flags: carry flag

Example: ASRD d2

CALL abs (call subroutine)

OP-Code: 0x11
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	0	0	0	1	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Call subroutine at absolute memory address.
The current program counter (3 bytes) is pushed to the MyCPU processor stack (not to the vCPU data stack).

h = bits 16-19 of target call address
m = bits 15-8 of target call address
l = bits 7-0 of target call address

Affected flags: none

Example: CALL 0x0438B3

CALL r (call subroutine via pointer)

OP-Code: 0x21
Size: 4 bytes

Encoding:

7							0	15							8
0	0	1	0	0	0	0	1	-	-	-	-	r	r	r	r

Call subroutine at address stored in register `r` (lower 16-bit part) and `r+1` (upper 16-bit part).
The current program counter (3 bytes) is pushed to the MyCPU processor stack (not to the vCPU data stack)..

r = register name r0-r15, e.g. r=8 for p0

Affected flags: none

Example: CALL p1

CIN r (console input)

OP-Code: 0x25
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	0	1	0	1	-	-	-	-	r	r	r	r

Read a character from the console standard input (e.g. a keyboard) into register `r`.
A zero is stored in register `r` if no key is pressed.

Note: In "Exclusive Access Mode" (page 10) the standard input is COM1

r = register name r0-r15

Affected flags: none

Example: CIN r0

CLRC (clear carry flag)

OP-Code: 0x08
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	1	0	0	0	-	-	-	-	-	-	-	-

Clears the carry flag.

Affected flags: carry flag

Example: CLRC

CLRI r (clear interrupt enable flag)

OP-Code: 0x0C
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	1	1	0	0	-	-	-	-	x	x	x	x

Copies the global interrupt enable flag into register x and sets it then to zero, so interrupts become globally disabled.

x = register name r0-r15

Affected flags: interrupt enable flag

Example: CLRI r1

CLRZ (clear zero flag)

OP-Code: 0x0A
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	1	0	1	0	-	-	-	-	-	-	-	-

Clears the zero flag.

Affected flags: zero flag

Example: CLRZ

CMPU r,r (compare unsigned)

OP-Code: 0x55
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	0	1	0	1	y	y	y	y	x	x	x	x

Compares the contents of register `x` with the contents of register `y`. The zero flag gets set when the compare matches, otherwise it is cleared. The carry flag gets set when `x` is equal or greater than `y`, otherwise the carry flag gets cleared.

x = register name r0-r15 of operand 1
y = register name r0-r15 of operand 2

Affected flags: zero and carry flag

Example: CMPU r4,r7

CMPU r,# (compare unsigned)

OP-Code: 0x75
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	0	1	0	1	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Compares the contents of register `r` with and an immediate value. The zero flag gets set when the compare matches, otherwise it is cleared. The carry flag gets set when `r` is equal or greater than the immediate value, otherwise the carry flag gets cleared.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: carry flag, zero flag

Example: CMPU r3,#0x0221

CMPUD r,r (compare unsigned, 32-bit)

OP-Code: 0x5D
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	1	1	0	1	y	y	y	y	x	x	x	x

Compares the contents of register `x,x+1` with the contents of register `y,y+1`. The zero flag gets set when the compare matches, otherwise it is cleared. The carry flag gets set when `x` is equal or greater than `y`, otherwise the carry flag gets cleared.

x = register name r0-r15 of operand 1, e.g. r=4 for d2
y = register name r0-r15 of operand 2, e.g. r=6 for d3

Affected flags: zero and carry flag

Example: CMPUD d2,d3

CMPS r,r (compare signed)

OP-Code: 0x56
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	0	1	1	0	y	y	y	y	x	x	x	x

Compares the contents of register `x` with the contents of register `y`. Before the compare is done, the value 0x8000 is added to the operands. The zero flag gets set when the compare matches, otherwise it is cleared. The carry flag gets set when `x` is equal or greater than `y`, otherwise the carry flag gets cleared.

x = register name r0-r15 of operand 1
y = register name r0-r15 of operand 2

Affected flags: zero and carry flag

Example: CMPS r4,r7

CMPS r,# (compare signed)

OP-Code: 0x76
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	0	1	1	0	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Compares the contents of register `r` with an immediate value. Before the compare is done, the value 0x8000 is added to the operands. The zero flag gets set when the compare matches, otherwise it is cleared. The carry flag gets set when `r` is equal or greater than the immediate value, otherwise the carry flag gets cleared.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: carry flag, zero flag

Example: CMPS r3,#0xA548

CMPSD r,r (compare signed, 32-bit)

OP-Code: 0x5E
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	1	1	1	0	y	y	y	y	x	x	x	x

Compares the contents of register `x,x+1` with the contents of register `y,y+1`. Before the compare is done, the value 0x8000 is added to the operands. The zero flag gets set when the compare matches, otherwise it is cleared. The carry flag gets set when `x` is equal or greater than `y`, otherwise the carry flag gets cleared.

x = register name r0-r15 of operand 1, e.g. r=4 for d2
y = register name r0-r15 of operand 2, e.g. r=6 for d3

Affected flags: zero and carry flag

Example: CMPSD d2,d3

COUT r (console output)

OP-Code: 0x26
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	0	1	1	0	-	-	-	-	r	r	r	r

Output a ASCII-character stored in register `r` to the standard output channel (e.g. a screen).

Note: In "Exclusive Access Mode" (page 10) the standard output is COM1

r = register name r0-r15

Affected flags: none

Example: COUT r0

DATA abs (skip data block)

OP-Code: 0x12
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	0	0	1	0	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Skip the following data block. This instruction can be seen as a relative jump behind the following data block. This instruction is useful to structure a program, so it becomes easier to disassemble it again.

h = bits 16-19 of data block size
m = bits 15-8 of data block size
l = bits 7-0 of data block size

Affected flags: none

Example: DATA 0x001200

DEC r,n (decrement)

OP-Code: 0x3D
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	1	1	1	0	1	c	c	c	c	r	r	r	r

Decrement the content of register `r` by count `c`. The zero flag gets set when the result of the decrement is zero, otherwise the zero flag is cleared.

r = register name r0-r15
c = count of decrements

Affected flags: zero flag

Example: DEC r3,1

DECD r,n (decrement, 32-bit)

OP-Code: 0x3F
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	1	1	1	1	1	c	c	c	c	r	r	r	r

Decrement the content of dword-register `r` and `r+1` by count c. The zero flag gets set when the result of the decrement is zero, otherwise the zero flag is cleared.

r = register name r0-r15, e.g. r=6 for d3
c = count of decrements

Affected flags: zero flag

Example: DECD d3,1

DIV r,r (divide)

OP-Code: 0x4B
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	1	0	1	1	y	y	y	y	x	x	x	x

Divide the content of register `x` by the content of register `y`. The result is stored in register `x`.

x = register name r0-r15 of the dividend and destination register
y = register name r0-r15 of the divisor

Affected flags: none

Example: DIV r2, r6 (r2 := r2 / r6)

DIVD r,r (divide, 32-bit)

OP-Code: 0x4C
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	1	1	0	0	y	y	y	y	x	x	x	x

Divide the content of register `x` and `x+1` by the content of register `y` and `y+1`. The result is stored in register `x` and `x+1`. The operands and the result are 32-bit wide.

x = register name r0-r15 of the dividend and destination register, e.g. r=2 for d1
y = register name r0-r15 of the divisor, e.g. r=6 for d3

Affected flags: none

Example: DIVD r2, r6 or DIVD d1,d3 (r2,r3 := r2,3 / r6,r7)

EXCL (exclusive access)

OP-Code: 0x07
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	0	1	1	1	-	-	-	-	-	-	-	-

Exclusive hardware access. After this OP-code was executed, the simulated vCPU gets exclusive access to the hardware of MyCPU. Background MyCPU services are no more executed, so the vCPU program is required to implement its own drivers for some of the MyCPU hardware.

After execution of this instruction all the available RAM memory is mapped for vCPU.
Note that after switching into "Exclusive Access Mode" (page 10) some OP-Codes are no more available or have a different function: GTS, HALT, SYS, EXCL, heap allocation

Affected flags: none

Example: EXCL

GTBV r,n (get transfer-buffer virtual address)

OP-Code: 0x2E
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	1	1	1	0	n	n	n	n	r	r	r	r

Get transfer buffer virtual 24-bit address into register `r` and `r+1`. The transfer buffer has a size of 4096 bytes. It is divided into slices of 256 byte. The parameter `n` can be seen as an offset of a multiple of 256 bytes into the transfer buffer. The virtual address that is returned by this op-code is calculated as "buffer_base_address + n * 256". The transfer buffer can be used to exchange data with the MyCPU kernel ROM.

Note: The transfer-buffers are not available in "Exclusive Access Mode" (page 10).

r = register name r0-r15, e.g. r12 for p2
n = buffer offset, e.g. 4 for 0x400 bytes

Affected flags: none

Example: GTBV p2,4

GTBP r,n (get transfer-buffer physical address)

OP-Code: 0x2F
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	1	1	1	1	n	n	n	n	r	r	r	r

Get transfer buffer physical 16-bit address into register `r`. The transfer buffer has a size of 4096 bytes. It is divided into slices of 256 byte. The parameter `n` can be seen as an offset of a multiple of 256 bytes into the transfer buffer. The physical address that is returned by this op-code is calculated as "buffer_base_address + n * 256". The transfer buffer can be used to exchange data with the MyCPU kernel ROM, and the physical address is used as parameter to kernel ROM calls.

Note: The transfer-buffers are not available in "Exclusive Access Mode" (page 10).

r = register name r0-r15
n = buffer offset, e.g. 4 for 0x400 bytes

Affected flags: none

Example: GTBP r0,4

GTS r (get timestamp)

OP-Code: 0x24
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	0	1	0	0	-	-	-	-	r	r	r	r

Get system timestamp into register `r`.
The timestamp counter is incremented at a rate of 30.51757813 Hz.

r = register name r0-r15

Affected flags: none

Example: GTS r6

Note:
When the vCPU operates in "Exclusive Access Mode" (page 10) the timer interrupt must be enabled, otherwise the timestamp counter does not increment.

HALT (stop program execution)

OP-Code: 0x06
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	0	1	1	0	-	-	-	-	-	-	-	-

Program execution is stopped. The vCPU interpreter gets quit, the program returns to the MyCPU command shell. The content of register r0 (bits 0-7) will be passed as exit-code to the shell.
In "Exclusive Access Mode" (page 10) this instruction will only halt the CPU.

Affected flags: none

Example: HALT

IDLE (execute background services)

OP-Code: 0x05
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	0	1	0	1	-	-	-	-	-	-	-	-

When your program is waiting for any event it should not only do busy-waiting. Instead you should insert this idle-instruction into your busy-waiting-loop. This allows the MyCPU-OS to execute background services.

Affected flags: none

Example: IDLE

IN r,abs (read I/O register)

OP-Code: 0x36
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	1	1	0	1	1	0	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Read a hardware I/O register (8 bit) into register `r`. The bits 15-8 of register `r` are set to zero.

h = bits 16-19 of the I/O address
l = bits 7-0 of the I/O address
r = target register name r0-r15

Affected flags: none

Example: IN r6,0x3E00

IN r,(r) (read I/O register)

OP-Code: 0x66
Size: 2 bytes

Encoding:

7							0	15							8
0	1	1	0	0	1	1	0	p	p	p	p	r	r	r	r

Read a hardware I/O register (8 bit) into register `r`, the bits 15-8 of register `r` are set to zero. The I/O register is addressed by the 16-bit value stored in pointer register p.

p = pointer register name r0-r15
r = target register name r0-r15

Affected flags: none

Example: IN r6,(r8)

INC r,n (increment)

OP-Code: 0x3C
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	1	1	1	0	0	c	c	c	c	r	r	r	r

Increment the content of register `r` by count `c`. The zero flag gets set when the result of the increment is zero, otherwise the zero flag is cleared.

r = register name r0-r15
c = count of increments

Affected flags: zero flag

Example: INC r3,1

INCD r,n (increment, 32-bit)

OP-Code: 0x3E
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	1	1	1	1	0	c	c	c	c	r	r	r	r

Increment the content of dword-register `r` and `r+1` by count `c`. The zero flag gets set when the result of the increment is zero, otherwise the zero flag is cleared.

r = register name r0-r15, e.g. r=4 for d2
c = count of increments

Affected flags: zero flag

Example: INCD d2,4

JPC abs (jump if carry)

OP-Code: 0x1A
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	1	0	1	0	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the carry flag is set.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: carry flag (not changed)

Example: JPC 0x064F29

JPEG abs (jump if result is equal or greater)

OP-Code: 0x1E
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	1	1	1	0	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the result of a compare is "equal or greater". A result is "equal or greater" when the carry flag is set or the zero flag is set.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: carry and zero flag (not changed)

Example: JPEG 0x064F29

JPEL abs (jump if result is equal or less)

OP-Code: 0x1F
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	1	1	1	1	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the result of a compare is "equal or less". A result is "equal or less" when the carry flag is cleared or the zero flag is set.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: carry and zero flag (not changed)

Example: JPEL 0x064F29

JPGR abs (jump if result is greater)

OP-Code: 0x1C
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	1	1	0	0	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the result of a compare is "greater". A result is "greater" when the carry flag is set and the zero flag is cleared.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: carry and zero flag (not changed)

Example: JPGR 0x064F29

JPLE abs (jump if result is less)

OP-Code: 0x1D
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	1	1	0	1	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the result of a compare is "less". A result is "less" when the carry flag is cleared and the zero flag is cleared.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: carry and zero flag (not changed)

Example: JPLE 0x064F29

JPNC abs (jump if not carry)

OP-Code: 0x1B
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	1	0	1	1	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the carry flag is not set.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: carry flag (not changed)

Example: JPNC 0x064F29

JPNZ abs (jump if not zero)

OP-Code: 0x19
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	1	0	0	1	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the zero flag is not set.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: zero flag (not changed)

Example: JPNZ 0x064F29

JPNZ r,abs (jump if register is not zero)

OP-Code: 0x14
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	0	1	0	0	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the register `r` is not zero.

r = register name r0-r15
h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: none

Example: JPNZ r5,0x064F29

JPZ abs (jump if zero)

OP-Code: 0x18
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	1	0	0	0	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the zero flag is set.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: zero flag (not changed)

Example: JPZ 0x064F29

JPZ r,abs (jump if register is zero)

OP-Code: 0x13
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	0	0	1	1	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address when the register `r` is zero.

r = register name r0-r15
h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: none

Example: JPZ r5,0x064F29

JUMP abs (jump absolute)

OP-Code: 0x10
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	0	0	0	0	h	h	h	h	-	-	-	-	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Jump to absolute memory address.

h = bits 16-19 of target address
m = bits 15-8 of target address
l = bits 7-0 of target address

Affected flags: none

Example: JUMP 0x064F29

JUMP r (jump to pointer)

OP-Code: 0x20
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	0	0	0	0	-	-	-	-	r	r	r	r

Jump to address stored in register `r` (lower 16-bit) and `r+1` (upper 16-bit)

r = register name r0-r15, e.g. r=8 for p0

Affected flags: none

Example: JUMP p2

LD r,# (load register)

OP-Code: 0x77
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	0	1	1	1	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Load an immediate value into register `r`.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: none

Example: LD r0,#0x1976

LDB r,abs (load byte from memory)

OP-Code: 0x30
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	1	1	0	0	0	0	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Load a byte from memory into register `r`. Only the lower 8 bits of the register get set, the upper bits are cleared to zero.

h = bits 16-19 of memory address
m = bits 15-8 of memory address
l = bits 7-0 of memory address
r = register name r0-r15

Affected flags: none

Example: LDB r9, 0x028E41

LDB r,(r) (load byte from memory)

OP-Code: 0x60
Size: 2 bytes

Encoding:

7							0	15							8
0	1	1	0	0	0	0	0	p	p	p	p	r	r	r	r

Load a byte from memory addressed by pointer register `p` into register `r`. Only the lower 8 bits of the register get set, the upper bits are cleared to zero.

p = pointer register name r0-r15, e.g. r=12 for p2
r = target register name r0-r15

Affected flags: none

Example: LDB r9,(p2)

LDB r,(r+ofs) (load byte from memory)

OP-Code: 0x68
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	0	1	0	0	0	p	p	p	p	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Load a byte from memory addressed by pointer register `p` and an offset into register `r`. The memory address is calculated by adding the unsigned 16-bit offset to the content of register `p`. Only the lower 8 bits of the register get loaded, the upper bits are cleared to zero.

h = bits 15-8 of the 16-bit offset
l = bits 7-0 of the 16-bit offset
p = pointer register name r0-r15, e.g. r=12 for p2
r = target register name r0-r15

Affected flags: none

Example: LDB r1,(p2+0x0218)

LDD r,abs (load dword from memory)

OP-Code: 0x34
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	1	1	0	1	0	0	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Load a double word (32-bit) from memory into register `r` and `r+1`.

h = bits 16-19 of memory address
m = bits 15-8 of memory address
l = bits 7-0 of memory address
r = register name r0-r15, e.g. r=6 for d3

Affected flags: none

Example: LDD d3, 0x028E41

LDD r,(r) (load dword from memory)

OP-Code: 0x64
Size: 2 bytes

Encoding:

7							0	15							8
0	1	1	0	0	1	0	0	p	p	p	p	r	r	r	r

Load a double word (32-bit) from memory addressed by pointer register `p` into register `r`.

p = pointer register name r0-r15, e.g. r=12 for p2
r = target register name r0-r15, e.g. r=6 for d3

Affected flags: none

Example: LDD d3,(p2)

LDD r,(r+ofs) (load dword from memory)

OP-Code: 0x6C
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	0	1	1	0	0	p	p	p	p	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Load a double word (32-bit) from memory addressed by pointer register `p` and an offset into register `r`. The memory address is calculated by adding the unsigned 16-bit offset to the content of register `p`.

h = bits 15-8 of the 16-bit offset
l = bits 7-0 of the 16-bit offset
p = pointer register name r0-r15, e.g. r=12 for p2
r = target register name r0-r15, e.g. r=2 for d1

Affected flags: none

Example: LDD d1,(p2+0x0218)

LDP r,# (load pointer)

OP-Code: 0x17
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	0	1	1	1	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Load a 20 bit memory pointer into register `r` and `r+1`.

h = bits 16-19 of memory address
m = bits 15-8 of memory address
l = bits 7-0 of memory address
r = register name r0-r15, e.g. r=8 for p0

Affected flags: none

Example: LDP p1, #0x07B382

LDW r,abs (load word from memory)

OP-Code: 0x31
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	1	1	0	0	0	1	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Load a word (16-bit) from memory into register `r`.

h = bits 16-19 of memory address
m = bits 15-8 of memory address
l = bits 7-0 of memory address
r = register name r0-r15

Affected flags: none

Example: LDW r1, 0x028E41

LDW r,(r) (load word from memory)

OP-Code: 0x61
Size: 2 bytes

Encoding:

7							0	15							8
0	1	1	0	0	0	0	1	p	p	p	p	r	r	r	r

Load a word (16-bit) from memory addressed by pointer register `p` into register `r`.

p = pointer register name r0-r15, e.g. r=12 for p2
r = target register name r0-r15

Affected flags: none

Example: LDW r9,(p2)

LDW r,(r+ofs) (load word from memory)

OP-Code: 0x69
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	0	1	0	0	1	p	p	p	p	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Load a word (16-bit) from memory addressed by pointer register `p` and an offset into register `r`. The memory address is calculated by adding the unsigned 16-bit offset to the content of register `p`.

h = bits 15-8 of the 16-bit offset
l = bits 7-0 of the 16-bit offset
p = pointer register name r0-r15, e.g. r=12 for p2
r = target register name r0-r15

Affected flags: none

Example: LDW r1,(p2+0x0218)

MCPY d,s,n (copy memory)

OP-Code: 0x7D
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	1	1	0	1	s	s	s	s	d	d	d	d	-	-	-	-	n	n	n	n	-	-	-	-	-	-	-	-

Copy a memory block. The register `s` and `s+1` is a pointer to the source memory. The register `d` and `d+1` is a pointer to the destination memory. The register `n` contains the number of bytes that shall be copied.

s = source pointer register name r0-r15, e.g. r2 for p1
d = destination pointer register name r0-r15, e.g. r6 for p3
n = number of bytes to copy, register name r0-r15

Affected flags: none

Example: MCPY p3,p1,r0

MOD r,r (modulo, 16-bit)

OP-Code: 0x46
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	0	1	1	0	y	y	y	y	x	x	x	x

The content of register `x` is divided by the content of register `y`. The remainder of this operation is stored in register `x`. The operands and the result are 16-bit wide.

x = register name r0-r15 of the dividend and destination register
y = register name r0-r15 of the divisor

Affected flags: none

Note: The result is undefined when y (the divisor) is zero.

Example: MOD r2, r6 (r2 := r2 % r6)

MODD r,r (modulo, 32-bit)

OP-Code: 0x4E
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	1	1	1	0	y	y	y	y	x	x	x	x

The content of register `x` and `x+1` is divided by the content of register `y` and `y+1`. The remainder of this operation is stored in register `x` and `x+1`. The operands and the result are 32-bit wide.

x = register name r0-r15 of the dividend and destination register, e.g. r=2 for d1
y = register name r0-r15 of the divisor, e.g. r=6 for d3

Affected flags: none

Note: The result is undefined when y (the divisor) is zero.

Example: MODD r2, r6 or MODD d1,d3 (r2,r3 := r2,3 % r6,r7)

MOV r,r (move)

OP-Code: 0x40
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	0	0	0	0	y	y	y	y	x	x	x	x

Move content of register `y` into register `x`.

x = destination register name r0-r15
y = source register name r0-r15

Affected flags: none

Example: MOV r4, r5 (MOV rx, ry / r4 := r5)

MOVD r,r (move, 32-bit)

OP-Code: 0x45
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	0	1	0	1	y	y	y	y	x	x	x	x

Move content of register `y` into register `x` and content of register `y+1` into register `x+1.`

x = destination register name r0-r15, e.g. r=2 for d1
y = source register name r0-r15, e.g. r=6 for d3

Affected flags: none

Example: MOV d1, d3 (MOV rx, ry / d1 := d3)

MOVHL r,r (move high-byte to low-byte)

OP-Code: 0x42
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	0	0	1	0	y	y	y	y	x	x	x	x

Move bits 8-15 of register y into bits 0-7 of register x. The bits 8-15 in register x remain unchanged.

x = destination register name r0-r15
y = source register name r0-r15

Affected flags: none

Example: MOVHL r4, r5 (MOVHL rx, ry / r4[0...7] := r5[8...15])

MOVLH r,r (move low-byte to high-byte)

OP-Code: 0x41
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	0	0	0	1	y	y	y	y	x	x	x	x

Move bits 0-7 of register y into bits 8-15 of register x. The bits 0-7 in register x remain unchanged.

x = destination register name r0-r15
y = source register name r0-r15

Affected flags: none

Example: MOVLH r4, r5 (MOVL rx, ry / r4[8..15] := r5[7...0])

MSET d,v,n (set memory)

OP-Code: 0x7C
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	1	1	0	0	v	v	v	v	d	d	d	d	-	-	-	-	n	n	n	n	-	-	-	-	-	-	-	-

Fill memory with a constant value. The register `d` and `d+1` is a pointer to the beginning of the memory that shall be filled with the value stored in register `v`. Only the bits 0 to 7 of register `v` are used (mset performs a byte-wise fill operation). The register `n` contains the size in bytes of the memory area to be filled.

d = destination pointer, register name r0-r15, e.g. r6 for p3
n = fill length, register name r0-r15
v = fill value, register name r0-r15

Affected flags: none

Example: MSET p3,r0,r2

MUL r,r (multiply)

OP-Code: 0x48
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	1	0	0	0	y	y	y	y	x	x	x	x

Multiply the content of register `x` with the content of register `y`. The result is stored in register `x`.

x = register name r0-r15 of operand 1 and destination register
y = register name r0-r15 of operand 2

Affected flags: none

Example: MUL r2, r6 (r2 := r2 x r6)

MULD r,r (multiply, 32-bit)

OP-Code: 0x4A
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	1	0	1	0	y	y	y	y	x	x	x	x

Multiply the content of register `x` and `x+1` with the content of register `y` and `y+1`. The result is stored in register `x` and `x+1`. The operands and the result are 32-bit wide.

x = register name of r0-r15 operand 1 and destination register, e.g. r=2 for d1
y = register name of r0-r15 operand 2, e.g. r=6 for d3

Affected flags: none

Example: MULD r2, r6 or MULD d1,d3 (r2,r3 := r2,3 x r6,r7)

MULX r,r (multiply extended)

OP-Code: 0x49
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	1	0	0	1	y	y	y	y	x	x	x	x

Multiply the content of register `x` with the content of register `y`. The result is stored in register `x` and register `x+1`. The result is 32-bit wide.

x = register name of operand 1 and destination register
y = register name of operand 2

Affected flags: none

Example: MULX r2, r6 (r2,r3 := r2 x r6)

NOP (no operation)

OP-Code: 0x00
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-

No function.

Affected flags: none

Example: NOP

NOT (bit-wise inversion)

OP-Code: 0x22
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	0	0	1	0	-	-	-	-	r	r	r	r

Bit-wise inversion (one's complement) of the content of register `r`.

r = register name r0-r15

Affected flags: none

Example: NOT r2

NEG r (negation)

OP-Code: 0x23
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	0	0	1	1	-	-	-	-	r	r	r	r

Negation (two's complement) of the content of register `r`.

r = register name r0-r15

Affected flags: none

Example: NEG r4

OR r,r (logical OR)

OP-Code: 0x51
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	0	0	0	1	y	y	y	y	x	x	x	x

Does a logical OR of the contents of register `x` and register `y`. The result of the operation is stored in register `x`.

x = register name r0-r15 of operand 1 and destination register
y = register name r0-r15 of operand 2

Affected flags: none

Example: OR r4,r7

OR r,# (logical OR)

OP-Code: 0x71
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	0	0	0	1	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Does a logical OR of the contents of register `r` and an immediate value. The result is stored back in register `r`.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: none

Example: OR r3,#0x2104

ORD r,r (logical OR, 32-bit)

OP-Code: 0x59
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	1	0	0	1	y	y	y	y	x	x	x	x

Does a logical OR of the contents of register `x,x+1` and register `y,y+1`. The result of the operation is stored in register `x` and `x+1`.

x = register name r0-r15 of operand 1 and destination register, e.g. r=4 for d2
y = register name r0-r15 of operand 2, e.g. r=6 for d3

Affected flags: none

Example: ORD d2,d3

OUT r,abs (write I/O register)

OP-Code: 0x37
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	1	1	0	1	1	1	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Write the lower 8 bits of register `r` to a hardware I/O register.

h = bits 15-8 of the I/O address
l = bits 7-0 of the I/O address
r = source register name r0-r15

Affected flags: none

Example: OUT r7,0x1402

OUT r,(r) (write I/O register)

OP-Code: 0x67
Size: 2 bytes

Encoding:

7							0	15							8
0	1	1	0	0	1	1	1	p	p	p	p	r	r	r	r

Write the lower 8 bits of register `r` to a hardware I/O register. The I/O register is addressed by the 16-bit value stored in pointer register p.

p = pointer register name r0-r15
r = source register name r0-r15

Affected flags: none

Example: OUT r7,(r9)

POP r,r (pull registers from stack)

OP-Code: 0x44
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	0	1	0	0	y	y	y	y	x	x	x	x

Load registers x to y from stack.

x = first register to pull
y = last register to pull

Affected flags: none

Example: POP r0,r15 (pulls registers r0-r15), POP r4,r3 (pulls r4 first, then pulls r3)

PUSH r,r (push registers to stack)

OP-Code: 0x43
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	0	0	1	1	y	y	y	y	x	x	x	x

Push content of registers x to y to the stack. The first register that is pushed is register x, the last is register y. Because the stack grows downwards, register y becomes a lower memory address on stack than register x. After the push operation the stack pointer points to the last element pushed. Since the stack contains only data and no caller-return-addresses, it is easy to use the stack for exchanging function parameters in subroutine calls. For example, the caller can push the function parameters to the stack, and the called function simply cleans up the stack by adding the size of the pushed parameters to the stack pointer register.

x = first register to push
y = last register to push

Affected flags: none

Example: PUSH r0,r15 (pushes registers r0-r15), PUSH r4,r3 (pushes r4 first, then pushes r3)

Note: PUSH r4,r4 and then PUSH r5,r5 constructs the same stack frame like PUSH r4,r5

RET (return from subroutine)

OP-Code: 0x01
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	0	0	0	1	-	-	-	-	-	-	-	-

The program counter is restored from MyCPU processor stack. Program execution continues after the initiating CALL command.

Affected flags: none

Example: RET

RETI (return from interrupt)

OP-Code: 0x02
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	0	0	1	0	-	-	-	-	-	-	-	-

The program counter is restored from MyCPU processor stack. CPU registers are restored. Program execution continues at the position where the interrupt had interrupted the program flow. The interrupt enable flag gets set again.

Affected flags: Interrupt enable flag

Example: RETI

ROTL r,n (rotate left)

OP-Code: 0x3A
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	1	1	0	1	0	n	n	n	n	r	r	r	r

Rotate the content of register `r` left by `n` positions. The carry flag is shifted into the LSB of register `r`. The MSB is shifted into the carry flag.

r = register name r0-r15
n = number of positions to shift

Affected flags: carry flag

Example: ROTL r6,2

ROTR r,n (rotate right)

OP-Code: 0x3B
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	1	1	0	1	1	n	n	n	n	r	r	r	r

Rotate the content of register `r` right by `n` positions. The carry flag is shifted into the MSB of register `r`. The LSB is shifted into the carry flag.

r = register name r0-r15
n = number of positions to shift

Affected flags: carry flag

Example: ROTR r6,4

SBC r,r (subtract with carry)

OP-Code: 0x54
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	0	1	0	0	y	y	y	y	x	x	x	x

Subtracts the content of register y and the negated carry flag from the contents of register x. If the operation underflows the carry flag gets cleared, otherwise the carry flag gets set. The result of the operation is stored in register x. The zero flag gets cleared when the result is nonzero.

x = register name r0-r15 of operand 1 and destination register
y = register name r0-r15 of operand 2

Affected flags: carry flag, zero flag

Example: SBC r4,r7

SBC r,# (subtract with carry)

OP-Code: 0x74
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	0	1	0	0	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Subtracts an immediate value and the negated carry flag from the contents of register r. If the operation underflows the carry flag gets cleared, otherwise the carry flag gets set. The result of the operation is stored in register r. The zero flag gets cleared when the result is nonzero.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: carry flag, zero flag

Example: SBC r3,#0x9074

SBCD r,r (subtract with carry, 32-bit)

OP-Code: 0x5C
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	1	1	0	0	y	y	y	y	x	x	x	x

Subtracts the content of register `y,y+1` and the negated carry flag from the contents of register `x,x+1`. If the operation underflows the carry flag gets cleared, otherwise the carry flag gets set. The result of the operation is stored in register `x,x+1`. The zero flag gets cleared when the result is nonzero.

x = register name r0-r15 of operand 1 and destination register, e.g. r=4 for d2
y = register name r0-r15 of operand 2, e.g. r=6 for d3

Affected flags: carry flag, zero flag

Example: SBCD d2,d3

SC r,r,r,abs (system call)

OP-Code: 0x7F
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	1	1	1	1	y	y	y	y	x	x	x	x	-	-	-	-	z	z	z	z	a	a	a	a	a	a	a	a

This instruction performs a system call. The involved registers are `x`, `y` and `z`. The system call function is indexed by the absolute value `a`.

See chapter "System Call Functions" on page 78 for a list of supported functions.

x = register name r0-r15
y = register name r0-r15
z = register name r0-r15
a = index number of the system call

Affected flags: carry flag, zero flag

Example: SC r0,r1,r2,0x81

SCPY d,s,n (string copy)

OP-Code: 0x7B
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	1	0	1	1	s	s	s	s	d	d	d	d	-	-	-	-	n	n	n	n	-	-	-	-	-	-	-	-

Copy a zero-terminated string in memory. The register `s` and `s+1` is a pointer to the source string in memory. The register `d` and `d+1` is a pointer to the destination memory. Before this instruction is called, the register `n` must be set to the maximum number of bytes to copy. The instruction will stop copying the string after `n` characters. It will automatically insert the terminating zero after the last copied character, thus the destination memory must have a size of `n+1` bytes.

s = source pointer register name r0-r15, e.g. r2 for p1
d = destination pointer register name r0-r15, e.g. r6 for p3
n = maximum length register name r0-r15

Affected flags: none

Example: SCPY p3,p1,r0

SETC (set carry flag)

OP-Code: 0x09
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	1	0	0	1	-	-	-	-	-	-	-	-

Sets the carry flag.

Affected flags: carry flag

Example: SETC

SETI r (set interrupt enable flag)

OP-Code: 0x0D
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	1	1	0	1	-	-	-	-	x	x	x	x

Sets the interrupt enable flag to the value in register x. If x contains a 1, interrupts get globally enabled. But if x contains a 0, interrupts get disabled.

x = register name r0-r15

Affected flags: interrupt enable flag

Example: SETI r1

SETZ (set zero flag)

OP-Code: 0x0B
Size: 2 bytes

Encoding:

7							0	15							8
0	0	0	0	1	0	1	1	-	-	-	-	-	-	-	-

Sets the zero flag.

Affected flags: carry flag

Example: SETZ

SEXT r,# (sign extension)

OP-Code: 0x15
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	0	1	0	1	-	-	-	-	r	r	r	r	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m

This instruction does a sign extension to a 32-bit signed value. The registers `r` and `r+1` contain the value that shall be sign-extended to 32-bit. The sign bit and thus the real size of the input value is given by the mask `m`. Only one bit in this mask is allowed to be set, that is the sign bit: For example set the mask to 0x80 to extend an 8-bit char to a 16-bit short integer or a 32-bit long integer. Set the mask to 0x8000 to extend a 16-bit short integer to a 32-bit long integer. The result of this operation will be stored back in register `r` and `r+1`.

r = source and destination register r0-r15, e.g. r2 for d1
m = 16-bit mask, is usually set to 0x0080 or 0x8000

Affected flags: none

Example: SEXT d1,#0x8000

SFTL r,n (shift left)

OP-Code: 0x38
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	1	1	0	0	0	n	n	n	n	r	r	r	r

Shift the content of register `r` left by `n` positions. The last left most overflowing bit is shifted into the carry flag.

r = register name r0-r15
n = number of positions to shift

Affected flags: carry flag

Example: SFTL r6,2

SFTR r,n (shift right)

OP-Code: 0x39
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	1	1	0	0	1	n	n	n	n	r	r	r	r

Shift the content of register `r` right by `n` positions. The last right most overflowing bit is shifted into the carry flag.

r = register name r0-r15
n = number of positions to shift

Affected flags: carry flag

Example: SFTR r6,8

SLEN r,p (string length)

OP-Code: 0x47
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	0	0	1	1	1	p	p	p	p	r	r	r	r

Get the length of a zero-terminated string. The register `p` and `p+1` is a pointer to the string in memory. Before this instruction is called, the register `r` must be pre-initialized with the maximum expected length of the string, or with zero for an unlimited or unknown length. After the execution of this instruction the register `r` will contain the calculated length of the string.

r = register name r0-r15
p =register name r0-r15, e.g. r2 for p1

Affected flags: none

Example: ld r0,#0x0200; ldp p1,#string; SLEN r0,p1

SOUT r (console string output)

OP-Code: 0x27
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	0	1	1	1	-	-	-	-	r	r	r	r

Output a ASCII text string to the standard output channel (e.g. a screen). The registers `r` and `r+1` contain the memory pointer to the string. The string output stops at the first zero in the stream.

Note: In "Exclusive Access Mode" (page 10) the standard output is COM1

r = register name r0-r15, e.g. r=8 for p0

Affected flags: none

Example: SOUT p0

SOUT r,r (console string output with length)

OP-Code: 0x4F
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	0	1	1	1	n	n	n	n	r	r	r	r

Output a ASCII text string to the standard output channel (e.g. a screen). The registers `r` and `r+1` contain the memory pointer to the string, and the register `n` contains the number of chars to be printed. The string output is not stopped by a zero in the stream.

Note: In "Exclusive Access Mode" (page 10) the standard output is COM1

r = register name r0-r15, e.g. r=8 for p0
n = register name r0-r15

Affected flags: none

Example: SOUT p0, r1

STB r,abs (store byte to memory)

OP-Code: 0x32
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	1	1	0	0	1	0	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Store a byte from register `r` into memory. Only the lower 8 bits of the register are stored.

h = bits 16-19 of memory address
m = bits 15-8 of memory address
l = bits 7-0 of memory address
r = source register name r0-r15

Affected flags: none

Example: STB r6,0x021A32

STB r,(r) (store byte to memory)

OP-Code: 0x62
Size: 2 bytes

Encoding:

7							0	15							8
0	1	1	0	0	0	1	0	p	p	p	p	r	r	r	r

Store a byte from register `r` into memory addressed by pointer register `p`.

p = pointer register name r0-r15, e.g. r=12 for p2
r = source register name r0-r15

Affected flags: none

Example: STB r1,(p2)

STB r,(r+ofs) (store byte to memory)

OP-Code: 0x6A
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	0	1	0	1	0	p	p	p	p	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Store a byte from register `r` into memory addressed by pointer register `p` and an offset. The memory address is calculated by adding the unsigned 16-bit offset to the content of register `p`.

h = bits 15-8 of the 16-bit offset
l = bits 7-0 of the 16-bit offset
p = pointer register name r0-r15, e.g. r=12 for p2
r = source register name r0-r15

Affected flags: none

Example: STB r1,(p2+0x0218)

STD r,abs (store dword to memory)

OP-Code: 0x35
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	1	1	0	1	0	1	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Store a double word from register `r` and `r+1` into memory.

h = bits 16-19 of memory address
m = bits 15-8 of memory address
l = bits 7-0 of memory address
r = source register name r0-r15, e.g. r=2 for d1

Affected flags: none

Example: STD d1,0x021A32

STD r,(r) (store dword to memory)

OP-Code: 0x65
Size: 2 bytes

Encoding:

7							0	15							8
0	1	1	0	0	1	0	1	p	p	p	p	r	r	r	r

Store a double word from register `r` and `r+1` into memory addressed by pointer register `p`.

p = pointer register name r0-r15, e.g. r=12 for p2
r = source register name r0-r15, e.g. r=6 for d3

Affected flags: none

Example: STD d3,(p2)

STD r,(r+ofs) (store dword to memory)

OP-Code: 0x6D
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	0	1	1	0	1	p	p	p	p	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Store a double word from register `r` and `r+1` into memory addressed by pointer register `p` and an offset. The memory address is calculated by adding the unsigned 16-bit offset to the content of register `p`.

h = bits 15-8 of the 16-bit offset
l = bits 7-0 of the 16-bit offset
p = pointer register name r0-r15, e.g. r=14 for sp
r = source register name r0-r15, e.g. r=2 for d1

Affected flags: none

Example: STD d1,(sp+0x0218)

STK r,func (call-stack operation)

OP-Code: 0x2D
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	1	1	0	1	f	f	f	f	r	r	r	r

Execute a special call-stack function.

r = register name r0-r15, e.g. r=2 for d1
f = function to execute

STK Functions:

f = 0 : Get callstack-pointer (current context) into r and r+1
f = 1 : Set callstack-pointer (current context). The registers r and r+1 must contain the new context.
f = 2 : Pull last call return-address from stack. The address is returned in r and r+1.
f = 3 : Push a return-address to stack. The registers r and r+1 must contain the address.

Notes:

1. It is not allowed to do operations on the callstack-pointer returned by function 0. This is because it is not really a “pointer” but rather a 32-bit “handle” that is returned.
2. The size of the call-stack is limited to 256 bytes (MyCPU processor callstack)
3. It is planned to used the STK-functions also to switch processor context between several “processes”. This will allow users to implement their own preemptive multitasking operating system on MyCPU. Due to several technical difficulties context-switching is not yet implemented.

STW r,abs (store word to memory)

OP-Code: 0x34
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	1	1	0	1	0	0	h	h	h	h	r	r	r	r	l	l	l	l	l	l	l	l	m	m	m	m	m	m	m	m

Store content of register `r` into memory.

h = bits 16-19 of memory address
m = bits 15-8 of memory address
l = bits 7-0 of memory address
r = source register name r0-r15

Affected flags: none

Example: STW r5,0x021A32

STW r,(r) (store word to memory)

OP-Code: 0x63
Size: 2 bytes

Encoding:

7							0	15							8
0	1	1	0	0	0	1	1	p	p	p	p	r	r	r	r

Store content of register `r` into memory addressed by pointer register `p`.

p = pointer register name r0-r15, e.g. r=12 for p2
r = source register name r0-r15

Affected flags: none

Example: STW r1,(p2)

STW r,(r+ofs) (store word to memory)

OP-Code: 0x6B
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	0	1	0	1	1	p	p	p	p	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Store content of register `r` into memory addressed by pointer register `p` and an offset. The memory address is calculated by adding the unsigned 16-bit offset to the content of register `p`.

h = bits 15-8 of the 16-bit offset
l = bits 7-0 of the 16-bit offset
p = pointer register name r0-r15, e.g. r=14 for sp
r = source register name r0-r15

Affected flags: none

Example: STW r4,(sp+0x0218)

SUB r,# (subtract)

OP-Code: 0x79
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	1	0	0	1	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Subtracts an immediate unsigned value from the contents of register `r`. If the operation underflows the carry flag gets cleared, otherwise the carry flag gets set. The result of the operation is stored in register `r`. The zero flag gets set when the result is zero, and it gets cleared when the result is nonzero.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: carry flag, zero flag

Example: SUB r7,#0x1561

SUBD r,# (subtract, 32-bit)

OP-Code: 0x6F
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	0	1	1	1	1	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Subtracts an immediate unsigned value from the contents of register `r` and `r+1`. If the operation underflows the carry flag gets cleared, otherwise the carry flag gets set. The result of the operation is stored in register `r` and r`+1`. The zero flag gets set when the result is zero, and it gets cleared when the result is nonzero.

r = register name r0-r15, e.g. r=6 for d3
h = bits 15-8 of the immediate value
l = bits 7-0 of the immediate value

Affected flags: carry flag, zero flag

Example: SUBD d3,#0x4B25

SYS r,r,abs (mycpu kernel call)

OP-Code: 0x7E
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	1	1	1	0	y	y	y	y	x	x	x	x	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

This instruction performs a MyCPU kernel function call. The address of the kernel function is given as absolute value in the instruction bits 16 to 31. Before this instruction can be executed, the registers x and y should be set to the appropriate values that are required by the called kernel function. The lower 8 bits of register x are copied into the mycpu-accumulator, and the content of register y is moved into the mycpu index registers X (bits 0-7) and Y (bits 8-15). Note that some kernel functions are ambiguous. To select one of the two sub-functions, you must clear or set the carry flag with the CLRC or SETC instruction. When the mycpu kernel call returns, the registers x and y are updated accordingly to the returned values of the kernel call, and the carry flag gets also updated. This instruction always clears the zero-flag (TBD).

Note:
The mycpu kernel call is performed by the indirect call instruction, op-code 0x1B. So only kernel vectors that are stored in RAM can be called.

x = register name r0-r15 (contains mycpu accumulator)
y = register name r0-r15 (contains mycpu x+y register)
h = bits 15-8 of the kernel call address
l = bits 7-0 of the kernel call address

Affected flags: carry flag, zero flag

Example: SYS r2,r3,0x0244

Note: This OP-Code is not available in "Exclusive Access Mode" (page 10).

TST r,# (test register with mask)

OP-Code: 0x16
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	0	0	1	0	1	1	0	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Test content of register `r` with a specified mask for zero. The zero flag gets set when the result of the AND operation with the register content and the supplied mask is zero.

r = register name r0-r15
h = bits 15-8 of the mask for the logical AND operation
l = bits 7-0 of the mask for the logical AND operation
Affected flags: Zero flag

Example: TST r1,#0x0400

TST r (test register)

OP-Code: 0x28
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	1	0	0	0	-	-	-	-	r	r	r	r

Test content of register `r` for zero. The zero flag gets set when the content of the register is zero.

r = register name r0-r15

Affected flags: Zero flag

Example: TST r1

TSTD r (test register, 32-bit)

OP-Code: 0x29
Size: 2 bytes

Encoding:

7							0	15							8
0	0	1	0	1	0	0	1	-	-	-	-	r	r	r	r

Test content of a double-word register `r` and `r+1` for zero. The zero flag gets set when the content of the register is zero.

r = register name r0-r15, e.g. r=2 for d1

Affected flags: Zero flag

Example: TSTD d1

XOR r,r (logical XOR)

OP-Code: 0x52
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	0	0	1	0	y	y	y	y	x	x	x	x

Does a logical XOR of the contents of register `x` and register `y`. The result of the operation is stored in register `x`.

x = register name r0-r15 of operand 1 and destination register
y = register name r0-r15 of operand 2

Affected flags: none

Example: XOR r4,r7

XOR r,# (logical XOR)

OP-Code: 0x72
Size: 4 bytes

Encoding:

7							0	15							8	23							16	31							24
0	1	1	1	0	0	1	0	-	-	-	-	r	r	r	r	l	l	l	l	l	l	l	l	h	h	h	h	h	h	h	h

Does a logical XOR of the contents of register `r` and an immediate value. The result is stored back in register `r`.

r = register name r0-r15
l = bits 7-0 of the immediate value
h = bits 15-8 of the immediate value

Affected flags: none

Example: XOR r6,#0x7FFF

XORD r,r (logical XOR, 32-bit)

OP-Code: 0x5A
Size: 2 bytes

Encoding:

7							0	15							8
0	1	0	1	1	0	1	0	y	y	y	y	x	x	x	x

Does a logical XOR of the contents of register `x,x+1` and register `y,y+1`. The result of the operation is stored in register `x` and `x+1`.

x = register name r0-r15 of operand 1 and destination register, e.g. r=4 for d2
y = register name r0-r15 of operand 2, e.g. r=6 for d3

Affected flags: none

Example: XORD d2,d3

System Call Functions

System calls are initiated with the "SC" instruction. The sc-instruction takes three registers as parameter, and an immediate value that denotes the function that shall be executed. This chapter contains a list with available functions.

A system call has the format

```
sc rx, ry, rz, fn
```

`rx`, `ry` and `rz` are the parameters to the system call, they can carry input values and/or are used to deliver result values of the call. `fn` denotes the function to be executed, it is a number in the range 0x00 - 0xFF.

0x80 - Allocate Heap Memory

Every call to this system function allocates one more 16kB-page of heap memory. This function does not take any input argument. When a new heap page was successfully allocated, the function returns the new end address of the heap in register `rx/rx+1`. In this case the carry flag is cleared. When no memory could be allocated the carry flag is set after this function call returns.
Note: This function is not available in "Exclusive Access Mode" (page 10).

`fn` = 0x80
`rx` = register that is used to return the new end address of the heap
`ry` = don't care
`rz` = don't care

Input flags : `C` = don't care, `Z` = don't care
Output flags : `C` = 0 on success, 1 if memory could not be allocated. `Z` = don't care

0x81 – Install Interrupt Handler

With the help of this function a user program can install its own handlers for hardware interrupts. In "Exclusive Access Mode" (page 10) all interrupts are available to the user program. In usual program mode only the timer interrupt is available.

Available Interrupts:

0 = Timer Interrupt, tick rate = 30.517578125 Hz
1 = Keyboard
2 = UART COM1
3 = UART COM2
4 = Ethernet Controller
5 - 7 = reserved

`fn` = 0x81
`rx` = interrupt number 0 - 7
`ry` = pointer to interrupt handler function (24-bit ptr, stored in `ry` and `ry+1`)
`rz` = don't care

Input flags : `C` = don't care, `Z` = don't care
Output flags : `C` and Z not changed

0x82 – Set Interrupt Enable Mask

This function is used to enable or disable hardware interrupts. The function takes one parameter in its rx register. The lower 8 bits contain a bit-mask of interrupts to enable. The upper 8 bits contain a bit-mask of interrupts to disable:

rx = (0x0100 << interrupt_to_disable) | (0x0001 << interrupt_to_enable)

For example, to enable the timer interrupt 0, set `rx` to 0x0001. If you want to disable the interrupt number 2, you must set `rx` to 0x0400. If you want to disable all interrupts, set `rx` to 0xFF00.

`fn` = 0x82
`rx` = mask of interrupts to enable/disable.
`ry` = don't care
`rz` = don't care

Input flags : `C` = don't care, `Z` = don't care
Output flags : `C` and Z not changed

0x83 – Convert String to PETSCII Format

This function converts a zero-terminated text string from ASCII format to PETSCII format. This is useful for exchanging strings with the MyCPU OS kernel.

`fn` = 0x83
`rx` = pointer to the string in one of the transfer buffers (rx must contain the physical address)
`ry` = maximum length of the string (the conversion will also stop at the first zero in the string)
`rz` = don't care

Input flags : `C` = don't care, `Z` = don't care
Output flags : `C` and Z not changed

0x84 – Convert String to ASCII Format

This function converts a zero-terminated text string from PETSCII format to ASCII format. This is useful for exchanging strings with the MyCPU OS kernel.

`fn` = 0x84
`rx` = pointer to the string in one of the transfer buffers (rx must contain the physical address)
`ry` = maximum length of the string (the conversion will also stop at the first zero in the string)
`rz` = don't care

Input flags : `C` = don't care, `Z` = don't care
Output flags : `C` and Z not changed

0x85 – Memory Compare

This function compares two memory regions with each other. It is a fast implementation of the C-function memcmp().

`fn` = 0x85
`rx` = pointer to the first memory region (rx,rx+1 = virtual 32-bit address)
`ry` = pointer to the second memory region (ry,ry+1 = virtual 32-bit address)
`rz` = length of the memory region

Input flags : `C` = don't care, `Z` = don't care
Output flags : `C` and Z are set accordingly to the result, so the conditional branch instructions can be used for equal / greater-than / less-than compares (Z is set when the result of the compare is zero (=equal), C is set when the result is equal or negative).

0x86 – String Compare (case sensitive)

This function compares two zero-terminated text strings with each other. It is a fast implementation of the C-function strncmp().

`fn` = 0x86
`rx` = pointer to the first text string (rx,rx+1 = virtual 32-bit address)
`ry` = pointer to the second text string (ry,ry+1 = virtual 32-bit address)
`rz` = maximum number of characters to compare

Input flags : `C` = don't care, `Z` = don't care
Output flags : `C` and Z are set accordingly to the result, so the conditional branch instructions can be used for equal / greater-than / less-than compares (Z is set when the result of the compare is zero (=equal), C is set when the result is equal or negative).

0x87 – String Compare (case insensitive)

This function compares two zero-terminated text strings with each other. It is a fast implementation of the C-function strnicmp().

`fn` = 0x87
`rx` = pointer to the first text string (rx,rx+1 = virtual 32-bit address)
`ry` = pointer to the second text string (ry,ry+1 = virtual 32-bit address)
`rz` = maximum number of characters to compare

Input flags : `C` = don't care, `Z` = don't care
Output flags : `C` and Z are set accordingly to the result, so the conditional branch instructions can be used for equal / greater-than / less-than compares (Z is set when the result of the compare is zero (=equal), C is set when the result is equal or negative).