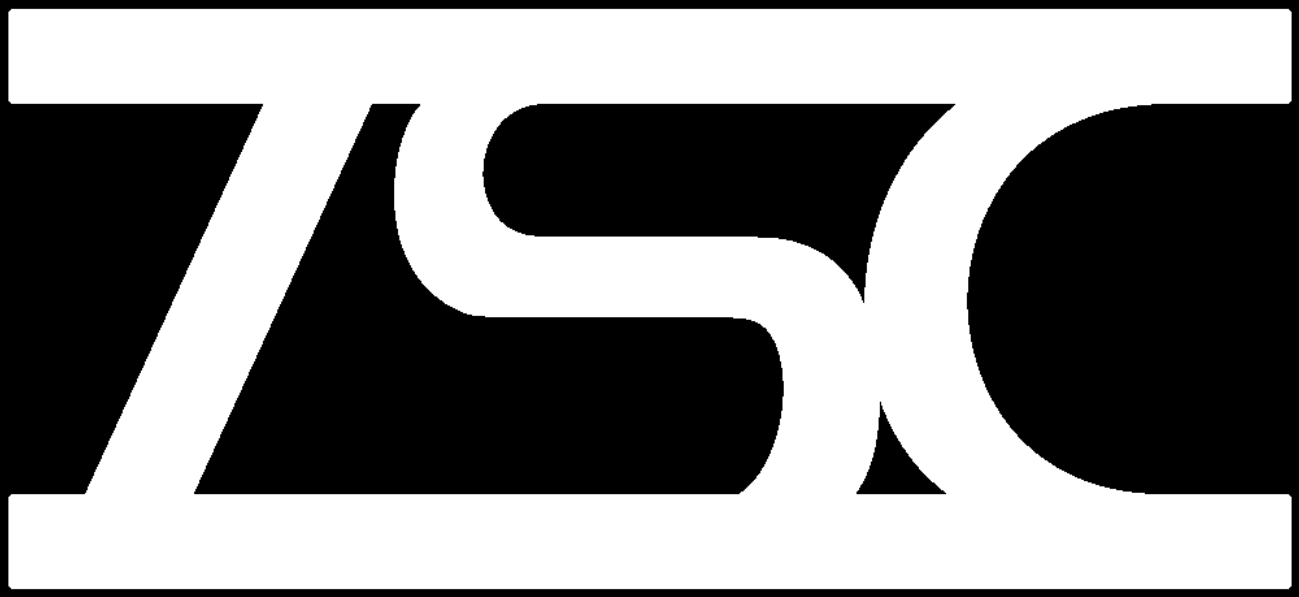


MICROCOMPUTERS

MICROCOMPUTERS

MICROCOMPUTERS

MICROCOMPUTERS

LWC33
HARDWARE & PROGRAMMING MANUAL

TURNER SEMICONDUCTOR

LWC33 Hardware & Programming Manual

Revision 2

26/ 03 / 2025

LWC33 Features

- Sixteen bit parallel processing
- 16 instructions
- 16 ALU Operations
- 12 General purpose registers
- 7 addressing modes
- True indexing capabilities
- Programmable stack pointer
- Variable length stack
- 16 bit bi-directional device bus
- Interrupt capability (4)
- Use with any type or speed memory
- 16 bit bi-directional Data Bus
- Addressable memory range of up to 1M words, 65K addressable at once.
- Sync output
- Pause pin

Pinout

Viewing the CPU from the top:

```
     IIIIIIIIIIIIIIII WR OOOOOOOOOOOOOOOO XXXX QQQQ

================================================

SSSS AAAAAAAAAAAAAAAA WR DDDDDDDDDDDDDDDD T YP C  V
```

I: Device ID output (Left to right: I0-I15)

W (top): Device bus write

R (top): Device bus read

O: Device bus data I/O (D0-D15)

X: AUX flag Set (X0-X3)

Q: Interrupt request (Q1-Q4)

S: Memory segment output (S0-S3)

A: Memory address output (A0-A15)

W (bottom): Memory write

R (bottom): Memory read

D: Memory data I/O (D0-D15)

T: Reset

Y: Sync out

P: Pause

C: Clock

V: Set carry

Device ID (D0-D15):

When reading or writing to a device, this is set to the requested device ID.

Device write (W top):

When writing to a device, this pin is active on the second half of the clock cycle.

Device read (R top):

When reading a device, this pin is active.

AUX flag set (X0-X3):

On the rising edge at any time, the AUX bit corresponding to the pin will be set.

Interrupt request (Q1-Q4):

While the CPU is fetching an instruction, the CPU will force load BRK when any of these pins are pulled HIGH, and the corresponding interrupt INT1-INT4 is executed.

Memory segment output (S0-S3):

Selects a memory segment to read/write.

Memory address output (A0-A15):

16 bit address to allow up to 65K of addressable memory at one time.

Memory write (W bottom):

Active on the second half of the clock cycle when writing to memory.

Memory read (R bottom):

Active while CPU is reading memory.

Memory data I/O (D0-D15):

16 bit bi-directional data bus, transferring data to and from memory.

Reset (T):

Reset is used to initialize the CPU, while this pin is held high, the CPU will not execute any instructions, the data bus is cleared, segment is cleared, and the address bus is set to 0xFFFF, and the read pin is pulled HIGH. On the falling edge of the next clock cycle after **reset** is LOW. The data currently on the memory data bus is loaded into the program counter. During a reset, interrupts are disabled, segment registers are cleared, V-MEM is disabled, but register states are not changed.

Sync out (S):

This pin is pulled HIGH while the CPU is fetching an instruction.

Pause (P):

After the CPU completes an instruction, if this pin is HIGH, CPU outputs (Segment, address, data, read, write) are cleared. The next clock cycle after pause is set to LOW again will resume normal execution.

Clock (C):

Clock input to the CPU.

Set carry (V):

Rising edge on this pin will set the carry bit in the status register.

Addressing modes

Immediate addressing:

This addressing mode is available to all instructions, the operand is contained in the second word of the instruction and no further memory addressing is required.

Absolute addressing:

Absolute addressing is only available to LOD (Load) and STO (Store) instructions. In this addressing mode, data is read/written to memory.

Indexed addressing:

Index addressing is only available to LOD and STO instructions. This addressing mode is used in conjunction with registers to read/write to memory at an indexed location. The effective address is calculated by adding the base address to the contents in a register.

Register addressing:

This form of addressing is represented with a one word instruction, implying an operation with one or more registers.

Implied addressing:

In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

Indirect addressing:

Indirect addressing, only available to the JMP (Jump) instruction. This addressing mode uses data read from memory as the effective address.

Relative addressing:

Relative addressing, only available to the JMP and JSR instructions. The effective address of relative addressing is determined by adding to the program counter. For example:

```
0x1000: BEQ 0x1005
```

Will result in an offset of 0x0004 (PC == 0x1001 when offset is applied)

Reset, fetch, execute, and interrupt cycles

Reset:

When the reset pin is pulled HIGH, the CPU will read from address 0xFFFF as the start vector to begin execution. The next clock cycle after the reset pin is LOW will latch the start vector to the PC.

Fetch:

During the fetch cycle, the CPU outputs the current program counter to the address bus and sets the segment to the code segment register, to read the next opcode. When the clock signal is pulled HIGH, the CPU latches the opcode into the instruction register, the PC is incremented, and the CPU starts to read the next value as the immediate data. When the clock goes back to LOW, the CPU latches the immediate data, the PC is incremented if immediate data is used in the opcode (otherwise the latched data is discarded), and the CPU is put into the execute phase. If the instruction does not do a memory access, the CPU will go back to fetch while executing the instruction.

Execute:

During the execute phase, the instruction is executed.

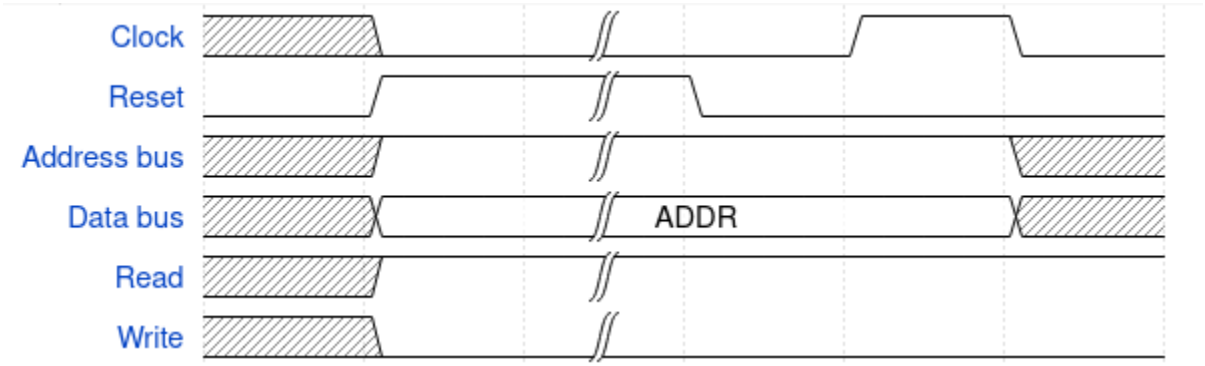
Interrupt:

The interrupt is triggered on the next fetch cycle when: The interrupt disable bit is cleared for the specified interrupt and the corresponding interrupt pin is pulled HIGH. When the interrupt is triggered, during the fetch cycle, the value 0x0X00 (BRK) is force loaded into the instruction register (where X is the interrupt number), but the PC is not incremented.

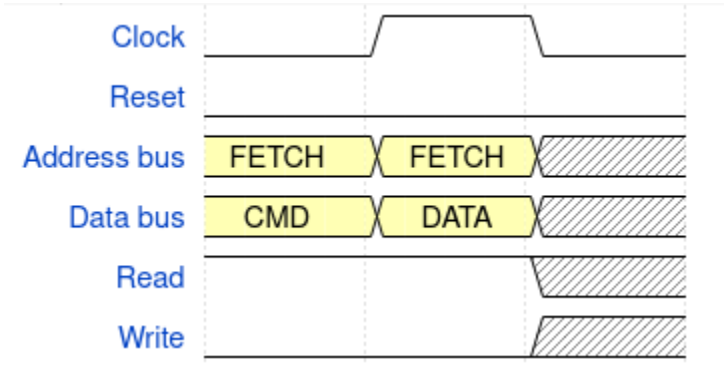
During an interrupt an interrupt is not able to trigger again until the CPU leaves the interrupt state with a return from interrupt (RTI) instruction. Exception being the BRK instruction able to trigger an interrupt while inside of an interrupt, if used improperly this can cause major bugs or a crash. If the interrupt is triggered by a BRK instruction, and not from the pin, the B bit is set in the status register.

Timings

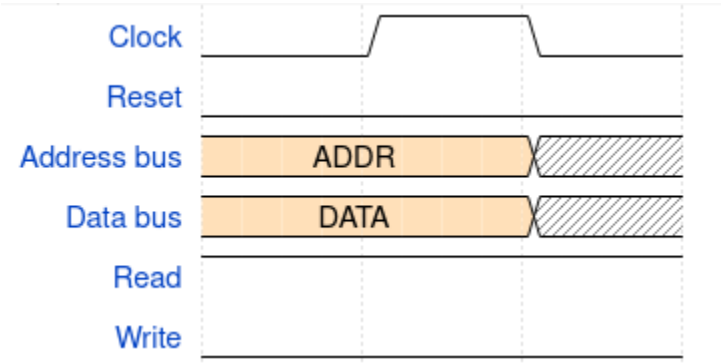
Reset cycle:



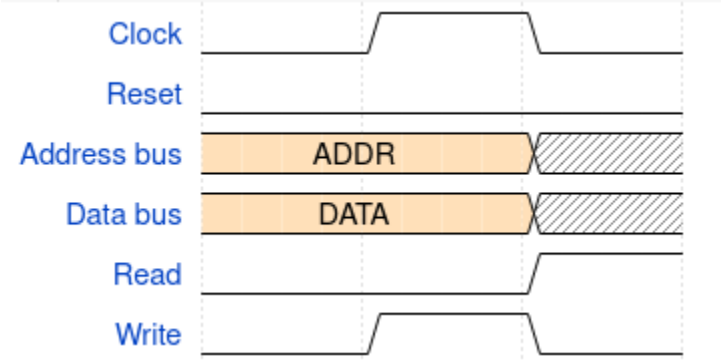
Fetch cycle:



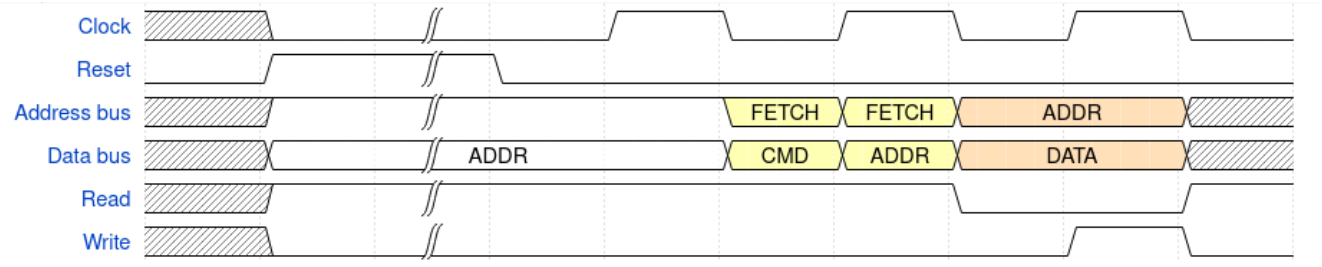
Execute cycle (Instruction that reads from memory):



Execute cycle (Instruction that writes to memory):



Full reset fetch and execute example (Instruction that writes to mem.):



PROGRAMMING

Instruction Set Architecture

Registers:

The LWC33 has 16 internal accessible registers. There are 12 general purpose registers and 4 special purpose registers.

Registers 1 to 12 are labeled r0-r11, register 13 is the result register (rr) (Modified using MUL, DIV, SMUL, SDIV and shifting operations, otherwise may be used like a normal register), register 14 is the stack pointer (sp), register 15 is a status register (st), and register 0 is a special immediate data, read-only register. Register 0 is latched with immediate data on the second phase of the fetch cycle, this register is used like any other register to provide a flexible, but simple instruction set.

In addition to the 16 main register, there are 16 sub-registers which can be read/written to using the SBR instruction.

Sub-register list:

0: INT0 - Interrupt 0 vector (This vector is used only with the BRK instruction)
1: INT1 - Interrupt 1 vector
2: INT2 - Interrupt 2 vector
3: INT3 - Interrupt 3 vector
4: INT4 - Interrupt 4 vector
5: BP - Base pointer
6: SM - Stack mask
7: VMV - Virtual memory base vector
8: VMS - Virtual memory size
9: VMSG - Virtual memory segment
10: VMBP - Virtual memory stack base pointer
11: VMSM - Virtual memory stack mask
12: CS - Code segment register
13: DS - Data segment register
14: SPSWP - Stack pointer swap, swapped when entering / exiting V-MEM mode
15: CTRL - General control register

Instruction encoding:

Instruction are encoded in 16 bit opcodes, the opcode contains the instruction, operand 1, operand 2, and operand 3.

IIIIXXXXYYYYZZZZ

Least significant bit is on the right.

I (4 bits): Instruction
X (4 bits): Operand 1 register selector
Y (4 bits): Operand 2 register selector
Z (4 bits): General instruction options / register selector (AKA Operand 3)

Each instruction may use one, two or all three operands as inputs. Operand 1 and 2 are selectors to select which registers to use as inputs to the instruction, instructions that generates a result will be stored back to the register selected with operand 1, except the memory write instructions where operand 1 is the register used to write to memory. Operand 3 is used differently depending on the instruction. Operand 3 may be used as the index register for store and load instructions, ALU function selector, conditional jump selector, etc.

Register (CONTINUED):

Stack pointer:

The stack pointer register is a read and writable register that is automatically incremented/decremented when using the stack instructions (push, pop, jump to subroutine, return from subroutine, and return from interrupt). The SP register is a pre-decrement / post-increment model, meaning that when pushing to the stack, the stack pointer is decremented before writing to memory, and reads from memory before incrementing.

The stack register does not point to a memory address directly. It is used in combination with the base pointer register for the final offset. When writing to this register, or when a stack operation is performed, it is bitwise anded together with the status mask register.

Base pointers:

The base pointer registers are the stack base location, when using the stack pointer in load/store instructions as the address or index, or when using a stack operation (push, pop, jump to subroutine, return from subroutine, and return from interrupt), the base pointer is added to the stack pointer as the effective address.

Stack mask:

The stack mask register is used to limit the stack's size, in powers of two. When writing to the stack pointer register, or when a stack operation is performed, the stack pointer is bitwise anded together with this register. This register must be set in a way to mask out upper bits in the stack pointer. For example, values like 0x000f, 0x001f 0xffff, are valid, where a value like 0x0505 is not.

Status register:

The status is a read and write register containing the flags that hold the CPU's state, in V-MEM mode the upper 8 bits cannot be changed. Status bits:

0: Carry (C) - Carry overflow from addition unit.
1: Zero (Z) - Set when the result from an instruction is zero.
2: Negative (N) - Set when the result from an instruction as the MSB set.
3: Unused
4: AUX0 - Set on the rising edge of X0
5: AUX1 - Set on the rising edge of X1
6: AUX2 - Set on the rising edge of X2
7: AUX3 - Set on the rising edge of X3
8: Interrupt disable 1 - Disables interrupt pin Q1
9: Interrupt disable 2 - Disables interrupt pin Q2
10: Interrupt disable 3 - Disables interrupt pin Q3
11: Interrupt disable 4 - Disables interrupt pin Q4
12: V-MEM enabled - Set while V-MEM mode is currently enabled, or used to enter V-MEM on a return from interrupt.
13: Absolute access - When set, programs running in V-MEM mode will be able to address all of memory instead only the virtual memory space.
14: Unsafe - Allows programs running in V-MEM mode to access devices above 0x00ff, modify the status register bits 8-15, modify any instruction, and modify any sub-register.

15: BRK (B) – Set when interrupt is triggered via BRK

Virtual memory register:

The virtual memory registers are used to setup the virtual memory. VMV (Virtual Memory Vector) points to the bottom of the virtual memory. VMS is the size of the virtual memory. VMSG is the segment which the virtual memory lives in, VMBP is the stack base pointer for virtual memory, VMSM is the stack mask for the virtual memory.

What does the virtual memory do? Virtual memory, like in it's name, is a virtual region of memory where the program running in it can only read, write and execute in that region. While in virtual memory mode the program is unable to write to sub-registers, write to the upper 8 bits of the status register, or access devices from 0x0100 to 0xffff (But can for devices 0x0000 to 0x00ff). For a program to access external functionality, it must use system calls, which can be achieved by using BRK to trigger INT0 with r0 set to a system call number, and other registers used as arguments.

Segment register:

The segment registers, CS (Code segment) and DS (Data segemnt), are use to select memory segments. The code segment is used whenever fetching instructions, and when reading/writing to memory (LOD, STO, and stack operations) the data segment is used.

Control sub-register:

The sub-register named control, or CTRL is used for some extra functionality. Currently when writing to this register with these bits active:

0: Backup general purpose registers and status register.

1: Restore general purpose registers and status register.

2: Restore general purpose registers only.

3: Restore status register only.

Instructions

The LWC33 has only 16 instructions, but due to the instruction encoding, every one of these instructions can be powerful.

0x0 – BRK

BReaK enters an interrupt routine. When this instruction is executed, the CPU pushes the current PC to the stack and sets the PC to the address set with the INT instruction. The B flag is set in the status.

Operand 1: Interrupt number

0x1 – JMP

JuMP sets the PC to the value in operand 1 if condition is true. Condition is set using bits, where if the specified bits set in the condition ANDed with the corresponding bits in the status register, result to a non-zero, the jump is taken. If invert is set, the jump is taken if the condition is false. For a jump always, set invert with no condition.

Operand 1: Address to jump to

Operand 2: Condition+Addressing selection

Bits:

0: AUX0 Condition bit

1: AUX1 Condition bit

2: Relative addressing

3: Indirect addressing

Operand 3: Condition

0: Carry conditional

1: Zero condition

2: Negative condition

3: Invert

0x2 – MOV

MOVe copies contents from one register to another.

Operand 1: Destination register

Operand 2: Source register

Operand 3: If set to 1, ZN is not updated

Modifies: ZN

0x3 – LOD

LoaD from memory at the specified address. If the stack pointer is used for addressing, the base pointer is automatically added to the effective address.

Operand 1: Destination register

Operand 2: Address

Operand 3: If not 0, this will specify an index register. (Effective address = OP2+OP3)

Modifies: ZN

0x4 – STO

STOre to memory at the specified address. If the stack pointer is used for addressing, the base pointer is automatically added to the effective address.

Operand 1: Source register

Operand 2: Address

Operand 3: If not 0, this will specify an index register. (Effective address = OP2+OP3)

0x5 – ALU

Perform an ALU operation on two registers and store the result back to the first operand. If the ALU function is bit shifting, operand 2 is ignored.

Operand 1: Data

Operand 2: Data

Operand 3: ALU function

Modifies: CZN (Does not modify the carry on bitwise operations and the shifting operations)

ALU functions:

0x0 – ADD (Add)

0x1 – ADC (Add with carry)

0x2 – SUB (Subtraction)

0x3 – SBC (Subtract with carry)

0x4 – AND (Bitwise AND)

0X5 – OR (Bitwise OR)

0x6 – XOR (Bitwise XOR)

0X7 – RAND (Gets a random number in the range from 0 to OP2 inclusive)

0x8 – SHL (Shift OP1 left by OP2 bits)

0x9 – ROL (Shift OP1 left by OP2 bits, bits shifted in are shifted from the RR register)

0xA – SHR (Shift OP1 right by OP2 bits)

0xB – ROR (Shift OP1 right by OP2 bits, bits shifted in are shifted from the RR register)

0xC – MUL (Unsigned multiply)

Upper 16 bits of result stored in rr

0xD – SMUL (Signed multiply)

Upper 16 bits of result stored in rr

0xE – DIV (Unsigned divide)

Remainder stored in rr

0xF – SDIV (Singed divide)

Remainder stored in rr

0x6 – CMP

CoMPare two values. Same as ALU, except that the result is not stored back into OP1.

Operand 1: Data

Operand 2: Data

Operand 3: ALU function

Modifies: CZN (Does not modify the carry on bitwise operations and the shifting operations)

0x7 – SBR

SuBRegister. This instruction is to read or write a sub-register.

Operand 1: Source/Destination register

Operand 2: Sub-register number

Operand 3: If 1, write register OP1 to sub-register, otherwise read sub-regsiter into OP1.

0x8 – JPS

JumP Segment jumps into a segment. This instruction sets the CS register and jumps to an address in that segment. This instruction also has a second function, which is to jump into virtual memory. When jumping into virtual memory, the interrupt state is also cleared, so this can be another way to exit the interrupt to virtual memory instead.

0x9 – DEV

DEVice. Read or write to a device through the device port.

Operand 1: Source/Destination register

Operand 2: Device number

Operand 3: If 1, write register OP1 to the device, otherwise read from the device into register OP1.

Modifies: ZN on read

0xA – PUSH

PUSH data onto the stack.

Operand 1: Data

0xB – POP

POP data off of the stack and write it into a register.

Operand 1: Destination register

Modifies: ZN

0xC – JSR

Jump to SubRoutine. Push the current PC to the stack and set the PC to operand 1. Operand 2 and operand 3 are the same as JMP, except JSR does not support indirect addressing.

Operand 1: Address

Operand 2: Condition+Addressing selection

Operand 3: Condition

0xD – RTS

ReTurn from Subroutine. Pop the return address from the stack and write it to the PC.

0xE – RTI

ReTurn from Interrupt. Pop the return address from the stack and write it to the PC, and exit from the interrupt state.

0xF – NOP

No Operation. This does nothing except waste a cycle.

ASSEMBLY REFERENCE

Letters in “supported classes” are what types of operands are valid for each instruction. Operand type list is listed under this table.

Mnemonic	Supported classes	Short Description
brk	B	Trigger interrupt
jmp	ABPQ	Jump to an address
jmp short	AB	Relative jump to address
jmp far	Da	Jump to address in segment
jmp virt	AB	Jump to address in vmem
jsr	AB	Jump to subroutine
jsr short	AB	Relative jump to subroutine
bcs	AB	Branch if carry set
bcc	AB	Branch if carry clear
beq	AB	Branch if zero/equal
bne	AB	Branch if not zero/eq.
bmi	AB	Branch if minus
bpl	AB	Branch if plus
bxa	AB	Branch if AUX0
bna	AB	Branch if not AUX0
bxb	AB	Branch if AUX1
bnb	AB	Branch if not AUX1
scs	AB	Subroutine if carry set (- Like bcs)
scc	AB	- Like bcc
seq	AB	- Like beq
sne	AB	- Like bne
smi	AB	- Like bmi
spl	AB	- Like bpl
sxa	AB	- Like bxa
sna	AB	- Like bna
sxb	AB	- Like bxb
snb	AB	- Like bnb
mov	CDFGHIJKLMNORST	Copy data between two locations

movs	CD	Copy data between registers without updating status
add	CDE	Add two values
adds	CDE	Add without storing
adc	CDE	Add with carry two values
adcs	CDE	Adc without storing
sub	CDE	Subtract two values
subs (Alias: cmp)	CDE	Sub without storing
sbc	CDE	Sub with carry
sbcs	CDE	Sbc without storing
and	CDE	And two values
ands (Alias: bit)	CDE	And without storing
or	CDE	Or two values
ors	CDE	Or without storing
xor	CDE	Xor two values
xors	CDE	Xor without storing
rand	CDE	Get random number
rands	CDE	Rand without storing
shl	CDE	Shift left value
shls	CDE	Shl without storing
rol	CDE	Rotate left value
rols	CDE	Rol without storing
shr	CDE	Shift right value
shrs	CDE	Shr without storing
ror	CDE	Rotate right value
rors	CDE	Ror without storing
mul	CDE	Multiply two values
muls	CDE	Mul without storing
smul	CDE	Multiply two signed values
smuls	CDE	Smul without storing
div	CDE	Divide two values
divs	CDE	Div without storing
sdiv	CDE	Divide two signed values
sdivs	CDE	Sdiv without storing
in	CD	Input data from device
out	CDE	Output data to device

push	AB	Push data to stack
pop	A	Pop data from stack
rts	0	Return from subroutine
rti	0	Return from interrupt
sec	0	Set carry
clc	0	Clear carry
sez	0	Set zero
clz	0	Clear zero
sen	0	Set negative
cln	0	Clear negative
sax0	0	Set AUX0
sax1	0	Set AUX1
sax2	0	Set AUX2
sax3	0	Set AUX3
clx0	0	Clear AUX0
clx1	0	Clear AUX1
clx2	0	Clear AUX2
clx3	0	Clear AUX3
sax	0	Set all AUX
clx	0	Clear all AUX
sei1	0	Set interrupt disable 1
sei2	0	Set interrupt disable 2
sei3	0	Set interrupt disable 3
sei4	0	Set interrupt disable 4
cli1	0	Clear interrupt disable 1
cli2	0	Clear interrupt disable 2
cli3	0	Clear interrupt disable 3
cli4	0	Clear interrupt disable 4
sei	0	Disable all interrupts
cli	0	Clear all interrupts
sevm	0	Set V-MEM enable
clvm	0	Clear V-MEM enable
saa	0	Set absolute access
caa	0	Clear absolute access
seu	0	Set unsafe

clu	0	Clear unsafe
backup	0	Backup all registers
restore	0	Restore all register
restoregp	0	Restore general purpose registers
restorest	0	Restore status register

Operand classes:

0: OPER
A: OPER REG
B: OPER U16/I16
C: OPER REG, REG
D: OPER REG, U16/I16
Da: OPER U16/I16, U4
E: OPER U16/I16, REG
F: OPER REG, [U16]
G: OPER REG, [REG]
H: OPER REG, [REG+U16]
I: OPER REG, [REG+REG]
J: OPER [U16], REG
K: OPER [REG], REG
L: OPER [REG], I16
M: OPER [U16+REG], REG
N: OPER [REG+REG], REG
O: OPER [REG+REG], I16
P: OPER (REG)
Q: OPER (U16)
R: OPER REG, SUBREG
S: OPER SUBREG, REG
T: OPER SUBREG, I16

MOV, MOVS – Copy value

```
mov r1, r2 ; Copy data from r2 into r1
mov r1, [addr] ; Load from address addr and store to r1
mov [addr], r1 ; Store r1 to address addr
mov r1, [addr+r2] ; Read from an indexed location
mov r1, [r2+r3] ; Read from an indexed location using only registers
```

mov simply copies the value from the second operand to the first. movs is a variation that doesn't update flags. To read or write memory, using square brackets define that the value should be used as a memory address. See supported classes F–O.

ADD, ADC, ADDS, ADCS, SUB, SBC, SUBS, SBCS, MUL, SMUL, DIV, SDIV, MULS, SMULS, DIVS, SDIVS – Arithmetic

```
add data, data
sub data, data
etc.
```

These instruction all perform arithmetic on the first and second operand, and the result is stores back to the first operand. The instructions with the "S" appended to the end does not store the value back to the first operand.

AND, OR XOR, ANDS, ORS, XORS – Bitwise logic

```
and data, data
or data, data
etc.
```

These instructions perform bitwise logic on the first and second operands, stores the result into the first operand. The instructions with the "S" appended to the end does not store the value back to the first operand.

SHL, SHR, ROL, ROR, SHLS, SHRS, ROLS, RORS – Shifting

```
shl r1, 8 ; Shift left by 8
rol r1, 8 ; Rotate left by 8
etc.
```

Shifting operands shift bits in operand 1 by the amount specified by operand 2. Before shifting, the value in op1 is copied into register **rr**. When performing a rotate instruction, the bits shifted in are from the data in register **rr**.

JMP, JMP SHORT, JMP FAR, JMP VIRT – Jumping

```
jmp addr ; Jump to absolute address addr
jmp (addr) ; Jump to address specific by data in memory at addr
jmp short addr ; Jump to addr using relative addressing
jmp far 0x8000, 4 ; Jump to address 0x8000 in segment 4
jmp virt 0x0000; Jump into virtual memory at address 0
```

Jumping changes the execution to the new address. **jmp** sets the program counter, **jmp short** sets the program counter relative to the current location, good for relocatable code. **jmp far** sets the code segment register and the program counter. **jmp virt** enables virtual memory mode and sets the program counter.

BCS, BCC, BEQ, BNE, BMI, BPL, BXA, BNA, BXB, BNB – Branching

```
bcs addr ; Branch if carry set to addr
bxa addr ; Branch if AUX0 flag is set
bne addr ; Branch if zero not set or compare not equals
etc.
```

Branching is a jump that happens only on the specified condition. Branching uses relative addressing.

JSR, JSR SHORT, RTS – Subroutines

```
jsr addr ; Execute subroutine at address addr
jsr short addr ; Execute subroutine using relative addressing
rts ; Return from subroutine
```

Subroutines are a piece of code which is able to return code execution back to where it was called from. When a **jsr** is executed, the current program counter is pushed to the stack automatically, and the program counter is set to the address of the subroutine. the **rti** instruction returns from the subroutine.

SCS, SCC, SEQ, SNE, SMI, SPL, SXA, SNA, SXB, SNB – Conditional subroutines

Conditional subroutines are almost exactly the same as normal branching, except that they do a jump to subroutine instead of a normal jump. Conditional subroutines are also relative addressing.

IN, OUT – Device I/O

```
in r1, 0x80 ; Read data from device 0x80 into r1
out r2, r1 ; Write data from r1 into device number stores in r2
```

The **in** and **out** instruction are used to read and write to devices on the CPU's device I/O port.