NIKTECH INC

Architecture Guide

NIKTECH INC

Architecture Guide

---

190 Shooting Star Isle
Foster City, CA - 94404

Table of Contents

About this guide.

This guide describes the Hardware architecture, Application Binary Interface and the instruction set of MANIK – a 32 bit RISC microprocessor.

Overview

MANIK is a 32 bit RISC Microprocessor. The salient features of the processor are listed below.

Features

Hardware Features

- Data Path Width 32 bits, with Four stage pipeline.
- Mixed 16/32 bit instructions for code density
- Von Neumann Architecture (Data and Instruction in the same address space).
- Sixteen, 32 bit General Purpose Registers.
- Four USER defined instructions (with Register File Write back capability).
- In-order issue Out-of-order completion.
- Some Conditional Instructions (Reduces branches & increases code density).
- Built in 32 bit Timer, (count down and interrupt modes)
- Power Down Mode.
- Multiplier and Barrel shifter (2 configurations, size Vs Performance trade off)
- Configurable Data & Instruction cache sizes.

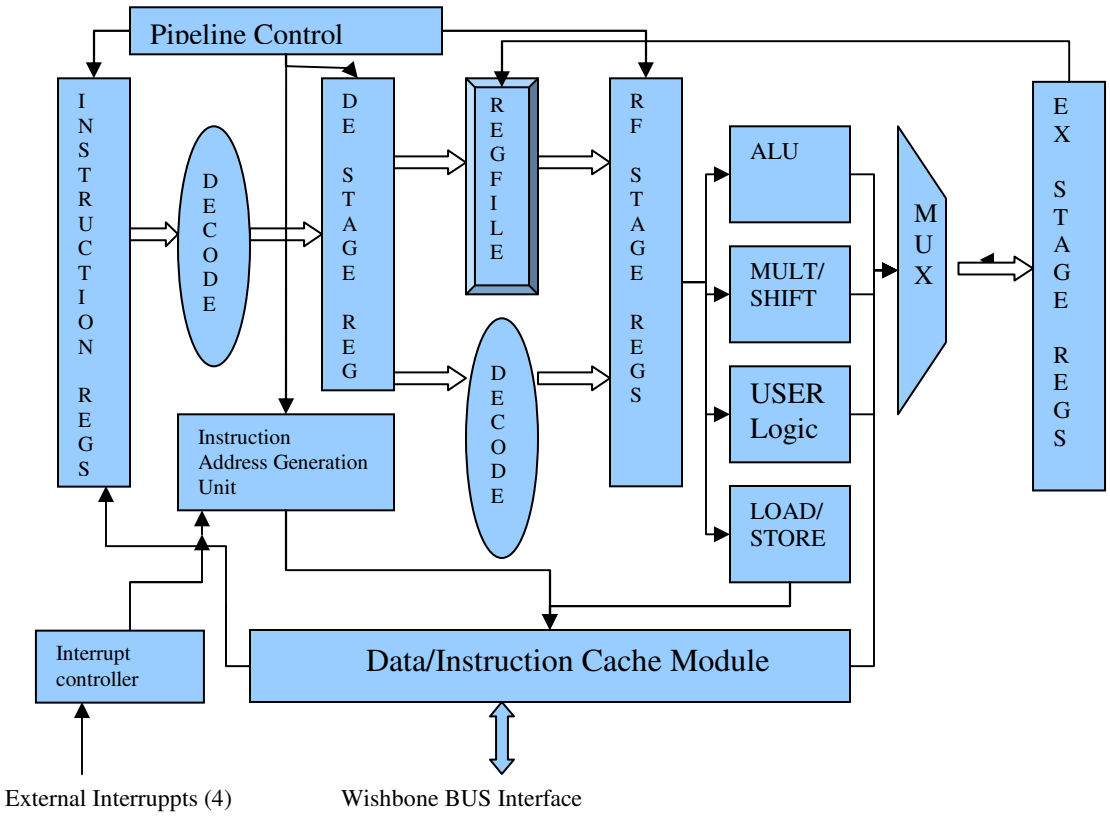
Wishbone bus compliant interface.

- Six External interrupts
- Hardware single step, Breakpoint & Watch point facility

Software Development Tools

- GNU Assembler, Linker (binutils)
- GCC (C,C++ Compiler)
- GDB (Debugger) and Instruction Set Simulator
- Standalone C-Library (RedHat newlib)

Block Diagram



Configuration Options

Option Name	***Default Value***	***Description***
WIDTH	32	Data Width (should not be changed)
ADDR_WIDTH	32	Address Width should not exceed Data Width
USREG_ENABLED	True	Enable USREG special function register
TIMER_WIDTH	32	Width of the built-in timer counter
TIMER_CLK_DIV	0	Timer counter decremented every ***this*** cycles
INTR_VECBASE	0	Configuration value **of** ibase **(**vector base)
INTR_SWIVEC	4	Offset of swi vector in the vector table
INTR_TMRVEC	8	Offset of Timer vector in the vector table
INTR_EXTVEC	12	Offset of external interrupt vector
INTR_BERVEC	16	Offset of Buserror interrupt vector
USER_INST	True	Logic for User defined instruction enabled

ICACHE_ENABLED	True	Instruction Cache enabled
DCACHE_ENABLED	True	Data Cache enabled
ICACHE_ADDR_WIDTH	10	Address of icache memory (determines icache size)
DCACHE_ADDR_WIDTH	10	Address of dcache memory (determines dcache size)
ICACHE_LINE_WORDS	1	Number of words(32 bit) per cache line (Number of read requests issued for a icache miss). Valid values (1,2,4,8)
DCACHE_LINE_WORDS	1	Number of words(32 bit) per cache line (Number of read requests issued for a dcache miss). Valid values (1,2,4,8)
ICACHE_SETS	1	Number of **icache** sets. Valid values (1,2,4)
DCACHE_SETS	1	Number of dcache sets. Valid values (1,2,4)
SHIFT_SWIDTH	3	Number of bits the barrel shifter will shift per cycle (max – 5, min – 1)
MULT_BWIDTH	32	Multiplier will take ***this****/32* cycles to complete. (max – 32, min – 2)
HW_BPENB	True	Enable 2 Hardware break points
HW_WPENB	True	Enable 2 Hardware watch points

Registers

There are sixteen 32-bit General Purpose registers in MANIK and four Special Function Registers.

General Purpose Registers

All sixteen General Purpose registers in MANIK are 32 bits wide, and are numbered from R0 to R15. The RESET value of all of them is UNDEFINED. The General Purpose Registers are orthogonal; there is no special or implied usage of any GPR by the hardware.

Register Name	*Compiler Usage*
R0	Stack Pointer
R1	1st Argument Register / Return Value
R2	2nd Argument Register / Return Value
R3	3rd Argument Register
R4	4th Argument Register
R5	Temporary register
R6	Temporary register
R7	Temporary register
R8	Register Variable (callee saved)
R9	Register Variable (callee saved)
R10	Register Variable (callee saved)
R11	Register Variable (callee saved)
R12	Register Variable (callee saved)
R13	Register Variable (callee saved)
R14	Register Variable (callee saved)
R15	Register Variable (callee saved)

Special Function Registers.

MANIK core has ***ten*** Special Function Registers. These registers can be read using the "mfsfr" instruction, and written to using the "mtsfr" instruction. The registers can be

referred to using their number (SFR0 – SFR10) or using their synonyms, PSW, RA, IPC, TIMER, IBASE, USREG, HWDBG, BP0, BP1, WP0 and WP1.

PSW (Program Status Word) (R/W) SFR0

Bit Position(s)	*Description*
0	SW – Soft Interrupt in progress
1	EI – External Interrupt in progress
2	TI – Timer Interrupt in progress
3	IP – Some interrupt in progress
4	TE – Enable timer interrupt
5	IE – Interrupt enable (global interrupt enable)
6	BIP– Backup IP flag. See General Interrupt handling for more details
7	PD – Power Down flag. See section on Power Down Mode for more details
8	TF – True false flag, 1 if last compare instruction is true. See Compare Instructions for more details
9	CY – Carry flag
10	BD – Flag Bypass Data Cache. See Cache operations for details.
11	II – Invalidate Instruction cache. See Cache Operations for details.
12	TR – Timer reload flag. See Section Timer for more details.(W/O)
12	TU – Timer underflow flag. See Section Timer for more details.(R/O)
13	SS – Single Step flag. See Section Single Stepping for more details
14	DBER – Data bus error, flag is set when a BUS error is detected during data load or store operation. See section Bus Error for more details.
15	IBER – Instruction bus Error, flag is set when a BUS error is detected during an instruction fetch. See section Bus Error for more details
19:16	SWI– Bits 3:0 of SWI instruction
20	EI0 – Enable External interrupt 0. See External Interrupt for details
21	EI1 – Enable External interrupt 1. See External Interrupt for details
22	EI2 – Enable External interrupt 2. See External Interrupt for details
23	EI3 – Enable External interrupt 3. See External Interrupt for details
24	EI4 – Enable External interrupt 4. See External Interrupt for details
25	EI5 – Enable External interrupt 5. See External Interrupt for details
26	ES0 – External interrupt 0 status. See External Interrupt for details
27	ES1 – External interrupt 1 status. See External Interrupt for details
28	ES2 – External interrupt 2 status. See External Interrupt for details
29	ES3 – External interrupt 3 status. See External Interrupt for details
30	ES4 – External interrupt 4 status. See External Interrupt for details
31	ES5 – External interrupt 5 status. See External Interrupt for details

RA (R/W) SFR1

Return address register, updated when a jump and link instruction is executed; it can also be updated using the "**mtsfr ra,Rn**" instruction , in this case the value is available after 2 cycles; there should not be a branch and link instruction within two instructions after a "**mtsfr ra, Rn**" instruction. See Jump Instructions for more details.

IPC (R/W) SFR2

Address of the instruction that would have executed when the core started processing an interrupt, the special function register can also be updated by software with the "**mtsfr ipc,Rn**" instruction.

TIMER (R/W) SFR3

Timer counter register, determines the frequency of timer interrupts see Timer for more details. Read returns the cycles remaining to zero.

IBASE(R/W) SFR4

This register determines the base address for the interrupt vector. The register can be configured with a value during system generation, or can be assigned a value by software, see section Interrupt handling for details.

USREG(R/W) SFR5

The register is available when the configuration parameter USREG_ENB is set to true. The register is not used by the processor and can be used a global register by the user application program.

HWDBG(R/W) SFR6

Hardware break/watch point control/status register.

Bit Position(s)	***Description***
0	BPENB (R/O)- Hardware break point is available. '1' when HW_BPENB = true
1	WPENB (R/O)- Hardware watch point is available. '1' when HW_WPENB = true
2	BP0_ENB (R/W)- enable BP0, the PC is compared with BP0 register
3	BP1_ENB (R/W)- enable BP1, the PC is compared with BP1 register
4	WP0_ENB (R/W)- enable WP0, the load/store address is compared with WP0 register
5	WP1_ENB (R/W)- enable WP0, the load/store address is compared with WP1 register
6	BP_HIT (R/O) – Hardware Break point is hit
7	WP0_HIT (R/O) – Watchpoint 0 is hit
8	WP1_HIT (R/O) – Watchpoint 1 is hit
9:31	(Reserved) – zeros

HWBP0(R/W) SFR7

Hardware Breakpoint address , available when HP_BPENB is true

HWBP1(R/W) SFR8

Hardware Breakpoint address , available when HP_BPENB is true

HWWP0(R/W) SFR9

Hardware Watchpoint address , available when HP_WPENB is true

HWWP1(R/W) SFR10

Hardware Watchpoint address , available when HP_WPENB is true

Instructions

Instruction Types

MANIK has six different types of instructions. Most of the instructions are 16 bits wide; the PC relative jump instructions are 32 bit wide. Following is a table of notations used in describing the instruction set.

Notation	Description
Rd	General Purpose Register operand R0 – R15
Rb	General Purpose Register operand R0 – R15
UIMM4	Unsigned 4 bit immediate operand [0..15]
SIMM4	Signed 4 bit immediate operand [-8..+7]
UIMM8	Unsigned 8 bit immediate operand [0..255]
SIMM8	Signed 8 bit immediate operand [-128..127]
SFRn	Special Function Register SFR0 – SFR3
EA	Effective address
SIMM27	Signed 27 bit immediate operand [-256 MB ... +256 MB]
sxt(x)	Sign Extend x to 32 bit
zxt(x)	Zero Extent x to 32 bit

ALU Instructions

All ALU instructions are executed by the arithmetic and logic unit and take one clock cycle to execute. MANIK has two formats for ALU instructions RR (Register, Register) and RI (Register, Immediate)

RR Form (Register, Register)

Syntax	15:12	11:8	7:4	3:0	Semantics
add Rd, Rb	0x2	0x1	Rd	Rb	Rd = Rd + Rb;
addc Rd, Rb	0x2	0x3	Rd	Rb	Rd = Rd + Rb + CY;
sub Rd, Rb	0x2	0x0	Rd	Rb	Rd = Rd – Rb;
subc Rd, Rb	0x2	0x2	Rd	Rd	Rd = Rd – Rb – CY;
mov Rd, Rb	0x2	0x4	Rd	Rb	Rd = Rb;
and Rd, Rb	0x2	0x5	Rd	Rb	Rd = Rd & Rb;
or Rd, Rb	0x2	0x6	Rd	Rb	Rd = Rd | Rb;
xor Rd, Rb	0x2	0x7	Rd	Rb	Rd = Rd ^ Rb;
sxb Rd, Rb	0x2	0x8	Rd	Rb	Rd[31:8] = Rb[7]; Rd[7:0] = Rb[7:0];
xhw Rd, Rb	0x2	0x9	Rd	Rb	Rd = (Rb << 16) | (Rb >> 16); Exchange HalfWord
sxh Rd, Rb	0x2	0xa	Rd	Rb	Rd[31:16] = Rb[15]; Rd[15:0] = Rb[15:0];
zxh Rd, Rb	0x2	0xb	Rd	Rb	Rd[31:16] = 0; Rd[15:0] = Rb[15:0];

RI Form (Register, Immediate)

Syntax	15:12	11:8	7:4	3:0	Semantics
andi Rd, UIMM8	0x5	UIMM8[7:4]	Rd	UIMM8[3:0]	Rd = Rd & zxt(UIMM8);
addi Rd, SIMM8	0x6	SIMM8[7:4]	Rd	SIMM8[3:0]	Rd = Rd + sxt(SIMM8);
movi Rd, SIMM8	0x7	SIMM8[7:4]	Rd	SIMM8[3:0]	Rd = sxt(SIMM8);

Load/store Instructions

MANIK can load and store data in 32 bit (word) , 16 bit (half word) or 8 bit (byte) chunks. The core aligns the address depending on the data size of the operation; four byte aligned for 32 bit (word) access (lower order 2 bits of address forced to zero); two byte aligned for 16 bit (half word) access (lowest order bit of address forced to zero). When loading data narrower than 32 bits , the upper (MS bits) of the destination register are set to zero, i.e. for half word load bits 31 - 16 of the destination register are set to zero, similarly for 8 bits loads bit positions 31 – 8 of the destination register are set to zero.

Syntax	15-12	11-8	7-4	3-0	Semantics

ldr Rd,uimm4(Rb)	0x8	UIMM4	Rd	Rb	EA = zxt(UIMM4 << 2) + Rb; Rd[31:0] = MEM[EA & 0xFFFFFFFC];
str Rd,uimm4(Rb)	0x9	UIMM4	Rd	Rb	EA = zxt(UIMM4 << 2)+ Rb; MEM[EA & 0xFFFFFFFC] = Rd;
ldrh Rd,uimm4(Rb)	0xa	UIMM4	Rd	Rb	EA = zxt(UIMM4 << 1)+ Rb; Rd[31:16] = 0, Rd[15:0] = MEM[EA & 0xFFFFFFFE];
strh Rd,uimm4(Rb)	0xb	UIMM4	Rd	Rb	EA = zxt(UIMM4 << 1)+ Rb; MEM[EA & 0xFFFFFFFFE] = Rd[15:0];
ldrb Rd,uimm4(Rb)	0xc	UIMM4	Rd	Rb	EA = zxt(UIMM4)+ Rb; Rd[31:8] = 0; Rd[7:0] = MEM[EA] ;
strb Rd,uimm4(Rb)	0xd	UIMM4	Rd	Rb	EA = zxt(UIMM4)+ Rb; MEM[EA] = Rd[7:0];

In addition to the above mentioned Register Indexed loads, the MANIK instruction set also provides a PC Relative load, this allows for loads of large constants that cannot be loaded into a register using the ALU instructions. The assembler generates a literal pool and puts address of LABEL in the literal pool; the offset of the entry in the literal pool is then inserted into the instruction. The LABEL can be replaced by a constant, in which case the assembler will put the constant value in the literal pool.

Syntax	*15:12*	*11:8*	*7:4*	*3:0*	*Semantics*
ldrpc Rd, LABEL	0x4	UIMM8[7:4]	Rd	UIMM8[3:0]	EA = PC + UIMM8*4; Rd = MEM[EA]

Jump Instructions

MANIK has three types of jump instructions a) ***short PC relative*** , b) ***long PC relative*** and c) ***Register indirect***. Short PC relative jump instructions are 16 bits wide and have a range of +/- 1KB, the Long PC relative jumps are 32 bit wide have a range of +/- 64MB; when the **sjt, sjf** or **sj** mnemonics are used the assembler determines the branch distance and generates the appropriate jump instruction, the **jt, jf** & **j** instruction forces the assembler to use **long** branch. Register indirect jump instructions are 16 bits wide. The register indirect form can use both General purpose registers and Special function registers as its source. The assembler inserts a NOP instruction after all "**jrl**" instruction to compensate for the size difference between the size of a "**jl**" and "**jrl**" instruction.

PC Relative Jumps

Syntax	*15:10*	*9:0*	*Semantics*
SJF Label	0xe4	SIMM10	If TF = 0 then PC += sxt(SIMM10*2);
SJT Label	0xed	SIMM10	If TF = 1 then PC += sxt(SIMM10*2);
SJ Label	0xf4	SIMM10	PC += sxt(SIMM27*2);
Syntax	*31:27*	*26:0*	*Semantics*
JF Label	0xe0	SIMM26	If TF = 0 then PC += sxt(SIMM26*2);
JT Label	0xe8	SIMM26	If TF = 1 then PC += sxt(SIMM26*2);
J Label	0xf0	SIMM26	PC += sxt(SIMM26*2);
JL Label	0xf8	SIMM26	RA = PC + 4; PC += sxt(SIMM26*2);

Register Indirect Jumps

Syntax	*15:8*	*7:4*	*3:0*	*Semantics*
JRL Rb	0x1c	0	Rb	RA = PC + 4; PC = Rb;
JR Rb	0x1d	0	Rb	PC = Rb;
JSFR SFRn	0x1e	0	SFRn	PC = SFRn

Multiply and Shift Instructions.

Multiply and shift operations are performed by a separate unit. The instructions a have two forms RR (Register, Register) and RI (Register Immediate). The Register Immediate form allows immediate values 0...15, to shift by more than 15 bits use the "xhw" instruction in combination with the shift instruction.

Syntax	*15:12*	*11:8*	*7:4*	*3:0*	*Semantics*
lsl Rd, Rb	0x2	0xc	Rd	Rb	Rd = Rd << Rb;
lsr Rd, Rb	0x2	0xd	Rd	Rb	Rd = Rd >> Rb; (Logical)
asr Rd, Rb	0x2	0xe	Rd	Rb	Rd = Rd >> Rb; (Arithmetic)
mult Rd, Rb	0x2	0xf	Rd	Rb	Rd = Rd * Rb;
lsli Rd, UIMM4	0x0	0xc	Rd	UIMM4	Rd = Rd << UIMM4;
lsri Rd, UIMM4	0x0	0xd	Rd	UIMM4	Rd = Rd >> UIMM4; (Logical)
asri Rd, UIMM4	0x0	0xe	Rd	UIMM4	Rd = Rd >> UIMM4; (Arithmetic)
multi Rd, UIMM4	0x0	0xf	Rd	UIMM4	Rd = Rd * UIMM4;

Compare Instructions

Compare instructions update the True/False flag (TF) in the PSW. These instructions will implicitly subtract the second operand from the first operand, and will set the True/False flag depending on the Carry, Overflow and Negative flag (Overflow and Negative flags are intermediate values and are not accessible by software). The compare instructions also have two forms RR (Register, Register) and RI (Register, Immediate); the RI form is only available for "equal-to" and the signed compares.

Syntax	*15:12*	*11:8*	*7:4*	*3:0*	*Semantics*
cmpeqi Rd, SIMM4	0x3	0x0	Rd	SIMM4	TF = (Rd == sxt(SIMM4))
cmpeq Rd, Rb	0x3	0x1	Rd	Rb	TF = (Rd == Rb)
cmphs Rd, Rb	0x3	0x2	Rd	Rb	TF = ((unsigned) Rd >= (unsigned) Rb)
cmpls Rd, Rb	0x3	0x3	Rd	Rb	TF = ((unsigned) Rd <= (unsigned) Rb)
cmplt Rd, Rb	0x3	0x4	Rd	Rb	TF = ((signed) Rd < (signed) Rb)
cmpgt Rd, Rb	0x3	0x5	Rd	Rb	TF = ((signed) Rd > (signed) Rb)
cmplti Rd, SIMM4	0x3	0x6	Rd	SIMM4	TF = ((signed) Rd < sxt(SIMM4))
cmpgti Rd, SIMM4	0x3	0x7	Rd	SIMM4	TF = ((signed) Rd > sxt(SIMM4))

Conditional instructions.

The MANIK instruction set provides conditional forms for some of the ALU instructions. The results of these instructions are written back to the register file depending on the value of the TF flag in the PSW.

Syntax	*15:12*	*11:8*	*7:4*	*3:0*	*Semantics*
addit Rd, SIMM4	0x3	0xe	Rd	SIMM4	If TF = 1 then Rd += sxt(SIMM4)
addif Rd, SIMM4	0x3	0xa	Rd	SIMM4	If TF = 0 then Rd += sxt(SIMM4)
movt Rd, Rb	0x3	0xc	Rd	Rb	If TF = 1 then Rd = Rb
movf Rd, Rb	0x3	0x8	Rd	Rb	If TF = 0 then Rd = Rb
addt Rd, Rb	0x3	0xf	Rd	Rb	If TF = 1 then Rd += Rb
addf Rd, Rb	0x3	0xb	Rd	Rb	If TF = 0 then Rd += Rb
subt Rd, Rb	0x3	0xd	Rd	Rb	If TF = 1 then Rd -= Rb
subf Rd, Rb	0x3	0x9	Rd	Rb	If TF = 0 then Rd -= Rb

User Defined Instructions.

MANIK allows four user defined instructions, each of these instructions can write back data to the register file. See USER Instruction interface for details.

Syntax	15:12	11:8	7:4	3:0	Sematics
udi0 Rd, Rb	0	0x8	Rd	Rb	Rd = User Defined Function (Rd, Rb)
udi1 Rd, Rb	0	0x9	Rd	Rb	Rd = User Defined Function (rd, Rb)
udi2 Rd, Rb	0	0xa	Rd	Rb	Rd = User Defined Function (rd, Rb)
udi3 Rd, Rb	0	0xb	Rd	Rb	Rd = User Defined Function (rd, Rb)

Pipeline.

The following sections describe the current pipeline implementation, it is likely to change in future releases of the processor core. The MANIK core has a four stage **bypassed** and **interlocked** pipeline. The pipeline stages are.

a) Instruction Fetch , Decode (IF)
b) Decode Stage (DE)
c) Register File Read (RF)
d) Execute (EX)

The execute (EX) stage consists of four units.

a) Arithmetic and Logic unit. This unit executes addition, subtraction and bitwise logic operations. All instructions executing in this unit has a *latency* of one cycle.
b) Multiply and Shift unit. This unit executes multiply and shift instructions, the MANIK pipeline can continue executing instructions following it in the pipeline as long they do not depend on the result of the multiply or shift instruction.
c) Load / Store unit, load and store instructions take two cycles when accessing data in the internal (on chip) storage, for external memory the number of cycles is variable. The core can continue executing instructions that do not depend on the result of the load instruction.
d) User defined unit. The interface to the User instruction is described in USER Instruction interface. All user instructions are assumed to write back to the register file, the pipeline will continue to execute instructions that do not depend on the result of the user instruction.

The pipeline will generally issue and retire one instruction per cycle; the order of retiring an instruction **may not be same as the order of issue**. In case a multi-cycle instruction is executing in the MULT/SHIFT, LOAD/STORE or User Defined Unit, instructions following it may proceed to completion if they do not depend on the result of the multi cycle instruction. If a dependent instruction is detected; i.e. an instruction that requires

the result of an incomplete multi-cycle instruction; the pipeline is stalled and no other instructions are allowed to proceed till the dependency has been resolved.

More than one multi cycle instruction may be executing simultaneously, e.g. a load instruction may be issued while a multiply /shift operation is in progress. More than one multi cycle instruction may complete in the same cycle; since the register file has only one write back port the pipeline is stalled till all the values are written back into the register file.

Branch Instructions take 2 cycles for a **taken** branch, 3 cycles for a **not-taken** branch. These latencies are when the code is resident in cache; if a cache miss occurs add the number of cycles taken to fetch the instruction from external memory into cache.

Multiply/Shift.

Multiply and Shift instructions are executed by the same unit. This unit has two configurations, a) for size and for b) for performance.

a) In the size optimized configuration multiply is performed 32x2 bits per clock cycle, it can take *up to* 16 cycles to complete. The shift operation is performed at the rate of one bit per clock cycle.
b) In the performance optimized configuration, the multiplication is performed with a 32x32 bit multiplier and takes 2 cycles to complete. The shifter is capable of shifting 8 bits per cycle; this implies that a shift operation can take between 2 to 5 cycles depending on the amount of shift. Shift amounts 0-7 completes in 2 cycles, 8-15 completes in 3 cycles, 16-23 takes 4 cycles, and shift amounts greater than 23 will take 5 cycles.

Cache operation.

MANIK has separate **data** & **instruction** caches. The data cache uses a **write through** policy. The cache sizes can be configured by changing the address line width. Each cache line can be configured to have 1,2,4 or 8 entries. The cache line size determines the number of 32 bit read requests a cache refill operation will perform. Increasing the cache line size will increase the size of the processor and can have adverse effect on the maximum frequency.
Both the data & instruction cache can be configured to have 1, 2 or 4 sets. The cache size is doubled or quadrupled when sets sizes of 2 or 4 chosen.

Although the instruction and data caches are separate, the MANIK core has **one Wishbone** master bus. In case of a simultaneous cache miss on the data and the instruction caches, the instruction cache is given priority over the data cache. Every store operation will write data into both the cache and the external memory.

The memory space is divided into two regions ***cacheable*** and ***I/O*** space. The size of the regions are configurable.

By default both the **Data** and **Instruction** cache are enabled. The data cache can be bypassed by setting the BD flag (bit position 10) in the PSW. The contents of the data cache are **not** preserved when the data cache bypass is activated.

The Instruction cache cannot be bypassed, specific instruction cache lines can be invalidated. This is done by setting the II Flag (bit position 11) in the PSW and performing a load or store operation; the instruction cache line corresponding to the address of the load or store will be invalidated. This facility can be used to download programs into memory as well as set break points, or any other situation that requires modifying code.

Fetching data or instructions from I/O space will bypass the corresponding cache.

Interrupt handling

MANIK has five interrupt vectors. The upper 24 Bits of the interrupt vector address is obtained from the **IBASE** (Special Function Register). The lower 8 bits can be configured during system generation the default values are listed below. By writing to the **IBASE** register the software could change the location of the interrupt vectors to a new location (useful during operating system boot process). The operation to complete a write to **ibase** register takes 2 cycles. The configuration value for the **ibase** can be set by the ***INTR_VECBASE***.

Vector Address	***Configuration Value***	***Description***	***PSW Bits effected***
0	-	Reset Vector	None
4	INTR_SWIVEC	Software/Single step instruction.	SS flag, SW flag, IP flag
8	INTR_TMRVEC	Timer Interrupt	TI flag, IP flag
12	INTR_EXTVEC	External Interrupt	EI0-EI5 flags
16	INTR_BERVEC	Bus Error	IBER and DBER Flags

When the processor recognizes an interrupt and branches to the **Interrupt vector** address, the IP flag (PSW bit 3) is copied to the BIP (PSW bit 6) flag and the IP flag is set to '1'. When the processor executes a "jsfr ipc" instruction, the BIP flag (PSW bit 6) is copied to the IP flag , and the BIP flag is set to zero. The BIP & IP flags can be updated by writing to the corresponding bit positions in the PSW.

The processor will branch to the interrupt service routine within 4 cycles of the detecting an interrupt. If there were instructions pending in the USER, Multiply/Shift or Load/Store units, they will continue to execute, the processor will stall if the Interrupt service routine accesses any of the registers being defined by any of the pending instructions.

The Timer & External interrupts are not recognized if the global interrupt enable flag IE (PSW bit 5) is zero, or if the processor is already servicing an interrupt; IP flag (PSW bit 3) is set.

The interrupt handler software is responsible for saving and restoring the PSW and any other registers it might use (including the Return Address register, RA (R/W) SFR1). Stack space can be created to save the registers can be created by using the "addi" instruction (this instruction does not update the PSW).

Nested Timer or External interrupts can be handled, by saving the IPC (R/W) SFR2, register on the stack, and re-enabling the interrupts by setting the IP & BIP flags in the PSW to zero.

Reset/Power up

On Reset / Power Up the processor will branch to the address 0 and start executing code from that address. This is a non-maskable interrupt. After Power up the values in General and Special purpose registers are undefined. The software is response for assigning initial values to the registers. The cache values remain unchanged after a reset interrupt.

Software Interrupt

When processor executes a SWI instruction, it executes a branch to address 0x4. The address of the next instruction is put into IPC (R/W) SFR2, register and the SW flag (bit position 0 in the PSW) is set to '1', the IP flag (bit position 3 in the PSW) is also set to '1'. It is recommended that the instruction **"swint 0x0f**" be used as a break point instruction. Bits 3:0 of the instruction are copied to bit positions 19:16 of the PSW.

The software interrupt ("swint") is NOT disabled by the IP flag, i.e. if an interrupt handler executes an "swint" instruction the processor will branch to the corresponding interrupt vector , and the IPC (R/W) SFR2, register will be updated. The TI & EI0-EI5 will retain their value upon return from the software interrupt. This allows the user to set breakpoints in the interrupt handler, the breakpoint should NOT be set before the interrupt service routine has had a chance to save the *registers* including the special function registers on the stack. Setting a breakpoint or single stepping in the software interrupt handler will have unpredictable results.

Timer Interrupt

The processor will jump to timer interrupt vector at address 0x8 when a Timer Interrupt occurs; see section Timer for more details. The address of the next instruction to be executed is stored in IPC (R/W) SFR2, and flags TI (PSW bit 2) and IP (PSW bit 3) are set to '1'. Note that the global interrupt enable bit IE (PSW bit 5) must be set for the timer to generate interrupts.

External Interrupt

MANIK Core has **six** External interrupt lines, each of the interrupt lines can be individually disabled by setting the EI0-EI5 bits in the PSW (bit positions 20 through 25); They can be collectively enable or disabled global interrupt enable flag, IE (bit position 5 in the PSW).

All the interrupts are level triggered, when the EXTRN_int line goes high the processor will jump to address 0x0c. The address of the next instruction to be executed is stored in IPC (R/W) SFR2, and flags EI (bit position 1 in the PSW) and IP (bit position 3 in the PSW) are set to '1', the External interrupt status flags ES0-ES5 (bit positions 26 through 31) are set depending on the interrupt being processed.
The interrupts are prioritized, with EI0 as the highest priority and interrupt EI5 as the lowest priority.

Bus Error Interrupt.

The core will branch to the Bus Error interrupt vector if the WBM_ERR_I (Wishbone bus error) signal is asserted during an instruction fetch or a data load/store operation. This is a non-maskable interrupt. The PSW bit 15 (IBER flag) is set if the exception occurred during an instruction fetch; the DBER flag (bit position 14 in the PSW) is set if the interrupt was generated during a load/store operation. This interrupt is an **imprecise** exception, i.e. the IPC register **may not** point to the instruction following the instruction that caused the exception, the IPC will point to the general vicinity of the instruction that caused the exception. It is recommended that the core do a software restart after a BUS error is signaled.

Hardware Debug aid

a) Hardware Single Stepping.

This facility is provided to help software debugging. When the **Single Step flag** in the PSW (Bit position 10) is set, the processor will generate an interrupt for every instruction executed, after it executes a **"jsfr ipc**" instruction. The Single step flag in the PSW would normally be set in an interrupt handling routine. The flag has no effect on execution till the "**jsfr ipc**" instruction is executed, the core will generate an interrupt when the instruction at the addresses pointed to by **ipc** is executed. The processor will branch to the same address as Software Interrupt, bit positions 23:16 of the PSW are all set to '1'. The software interrupt handler can determine the cause by examining bit position 0 in the PSW, a '1' in this field means it was invoked by a **swint** instruction, else it was entered due to a **single step** instruction.
The **IPC** register will contain the address of the instruction that would have executed next. External and Timer interrupts are disabled during the **Single Step** operation.

b) Hardware Breakpoints

The configuration option **HW_BPENB** enables two hardware breakpoint registers. This allows the debugger to place breakpoints in the application without modifying the code space. This is useful for setting breakpoints in **readonly** program area such as flash memory. Hardware breakpoints can be set using the following steps

a) Enable HW_BPENB configuration option
b) Set the address for the breakpoint in one or both of the two hardware breakpoint registers (BP0 and/or BP1)
c) Enable the corresponding bit(s) in the HWDBG register. BP0_ENB for BP0, or BP1_ENB for BP1.

When the RF stage PC matches one the BP registers (BP0 or BP1), and the corresponding enable bits are set in the HWDBG register, the processor core will branch to the address of Software Interrupt, bit positions 23:16 of the PSW are all set to '1'.The software interrupt handler can determine the cause of the interrupt by examining the BP_HIT (bit 6) in the HWDBG register.
Note the interrupt is generated **before** the instruction with the address match is executed. The IPC register will contain the address of the instruction.

c) Hardware Watchpoints

The configuration option **HW_WPENB** will enable two hardware watchpoints registers. The debugger can use these registers to be notified when a certain memory address is read(load) or written(store) to. The following steps are required to use this feature.

a) Enable HW_WPENB configuration option
b) Set the address(es) of the memory location in either or both of the watchpoint registers (WP0 and/or WP1).
c) Enable the corresponding bit(s) in the HWDBG register. WP0_ENB for WP0, or WP1_ENB for WP1.

The processor will compare the load/store address to the WP0 and/or WP1 register, if they match and the corresponding enable bit(s) (WP0_ENB and/or WP1_ENB) are set in the HWDBG register, the processor will branch to the address of Software Interrupt, bit positions 23:16 of the PSW are all set to '1'. The processor will also set bits WP0_HIT (if address in WP0 matched), and/or WP1_HIT (if address in WP0 matched). The software interrupt handler can use the WP0_HIT, WP1_HIT flags in the HWDBG registers to determine the cause of the interrupt.
Note that the interrupt is generated **after** the load/store instruction with the matching address is executed.

Timer.

The MANIK core includes a built in 32 bit timer. The Timer has two modes of operations.

a) **Interrupt Mode.** In this mode the Timer is started by enabling the TE flag (bit position 4 in the PSW) and by writing a value in the TIMER (R/W) SFR3 register. The timer is implemented as a down counter, the value written to the timer is copied to a register, this register is then decremented every clock ("coreclk") cycle. A Timer interrupt is generated when the value of this register becomes zero. Note the IE flag (bit position 5 in the PSW) must be set for the timer to generate interrupts.

The TIMER (R/W) SFR3, can be updated either in normal operation mode (IP flag is zero) or when the core is processing a timer interrupt (IP flag is one and TI flag is one). Reading the Timer register returns the current value. The timer interrupt ***can*** occur in Power Down Mode.

b) **Counter Mode.** In this mode the Timer register can be used to count the number of cycles. To operate the **Timer** in this mode an initial value must be written to the ***Timer*** register, then a '1' is written into the **TR** bit (position 12) in the PSW. The Timer will start the down counter, it will stop when the counter reaches zero. To determine the number of cycles executed the **Timer** register must be read and subtracted from the reload value. If the **TU** flag (Bit position 12) in the PSW is set to '1' then the there was an underflow, to get a more accurate cycle count the reload value should be increased. The largest reload value is 0xffffffff (4294967295 cycles).

Power Down Mode.

The processor can be put into a Power Down mode by setting the PD flag (bit position 7 in the PSW). In this mode the Timer & External interrupt are enabled but no instructions are fetched. On receiving an interrupt the processor will branch to the corresponding vector, IPC (R/W) SFR2 register will contain the address of the instruction following the "mtsfr PSW, Rn" instruction that caused the processor to go into Power Down mode. The following assembly language snippet illustrates the usage of Power Down Mode.

```
        .text
        .global _start
_start:
        j       main
        j       null_isr
        j       timer_isr
        j       null_isr
main:
        ldrpc   r0,_SP_START
        # timer expires after 16 cycles
        movi    r1,16
        mtsfr   timer,r1
        # enable timer & interrupt
        movi    r1,0x30
        mfsfr   r2,psw
        or      r2,r1
        mtsfr   psw,r2

stop:
        movi    r2,-1
        andi    r2,0x80
        mfsfr   r1,psw
        or      r1,r2
        mtsfr   psw,r1          # Set Power Down Flag
        sj      stop            # will come here when timer_isr finishes

timer_isr:
        addi    r0,-8
        str     r1,0(r0)
        str     r2,4(r0)
        # re enable timer interrupt
        mfsfr   r1,psw
        movi    r2,0x10
        or      r1,r2
        mtsfr   psw,r1
        ldr     r2,4(r0)
        ldr     r1,0(r0)
```

```
        addi    r0,8
null_isr:
        jsfr    ipc

_SP_END:
        .org    _SP_END+126
_SP_START:
```

USER Instruction interface.

The User defined extensions to the microprocessor can take two 32-bit values as input and write back one 32-bit value to the register file. The contents of the registers Rd, and Rb used in the user instruction are given as two operands to the User extension, the output value from the user extension is written back to the Rd register. The current implementation of the pipeline allows for the core to continue execution of other independent instructions, however only one User instruction can be outstanding at any time. Eight signals are used to interface the User extension to the core of the microprocessor; these are described in the following table. All signals are synchronized on clock signal "coreclk".

Name	Width (Bits)	Direction	Description
UINST_uiop	2	Out	Has values 0-3 corresponding to udi0-3
UINST_uinst	1	Out	1 indicates that UINST_uiop signal is valid. The operands may not be valid.
UINST_Nce	1	Out	The operand values are valid when '0'. The user extension should latch UINST_uiop and the operands when this signal is '0' and UINST_uinst is '1'.
UINST_uiopA	32	Out	1st operand .Value of Rd
UINST_uiopB	32	Out	2nd operand .Value of Rb
UINST_ip	1	In	Should be set to '1' when user core is busy. UINST_out is considered valid when this signal is low.
UINST_out	32	In	Value to be written back to Rd.
UINST_wbc	1	Out	UINST_out and UINST_ip signals should be held valid till this signal goes high. The core will set this signal to high when the value is written back to the register file.

The following VHDL snippet shows an example state machine which implements the USER interface.

```
Architecture sample of  UDI_sample is
    type udi_states is (S0, S1, ...  SWRITE_BACK, SLAST);
    attribute ENUM_ENCODING : string;
    attribute ENUM_ENCODING of udi_states : type is "000 001 010 011 . . . ";
    signal curr_udistate : udi_states := S0;
    signal opa_latch : std_logic_vector (WIDTH-1 downto 0) := (others => '0');
    signal opb_latch : std_logic_vector (WIDTH-1 downto 0) := (others => '0');

BEGIN
    -- simulate simple USER instruction
    udi_proc : process (coreclk)
    begin
        if rising_edge(coreclk) then
            case curr_udistate is
                when S0 =>
                -- idle then latch operands when we should
                    if UINST_uiop = "00" and UINST_uinst = '1' and
                       UINST_Nce = '0' then
                        opa_latch     <= UINST_uiopA;
                        opb_latch     <= UINST_uiopB;
                        UINST_uip     <= '1';
```

```
                    curr_udistate <= S1;
                else
                    curr_udistate <= S0;
                    UINST_uip     <= '0';
                end if;

            when S1 =>
                  ---
                  --- Do Processing WITH opa_latch & opb_latch
                  ---
                  ...
                  ...
            when SWRITE_BACK =>
                UINST_out     <= opa_latch;
                UINST_uip     <= '0';
                curr_udistate <= SLAST;

            when SLAST =>
                if UINST_wbc = '1' then
                    curr_udistate <= S0;
                else
                    curr_udistate <= SLAST;
                end if;
            when others => null;
        end case;
    end if;
end process udi_proc;
```

MANIK Wishbone Bus Interface.

MANIK uses the open standard Wishbone Bus to interface with external memory and I/O devices. The Wishbone bus specifications can be found at http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf. MANIK's bus interface signals are listed in the following table.

Signal Name	In/Out	Width	Description
WBM_DAT_I	In	32	Input data from the external device
WBM_ACK_I	In	1	Input data is valid
WBM_DAT_O	Out	32	Data output to the external device
WBM_SEL_O	Out	4	Byte lane select (see following table)
WBM_WE_O	Out	1	Write enable output
WBM_STB_O	Out	1	Write/Read Strobe High through entire cycle
WBM_CYC_O	Out	1	Valid bus Cycle in progress
WBM_LOCK_O	Out	1	Cycle in progress no arbitration allowed
WBM_ADR_O	Out	32	Address
WBM_CTI_O	Out	3	Cycle Type
WBM_BTE_O	Out	2	Burst Type

In the current implementation the Bus transactions generated, does not use the CTI_O and the BTE_O signals. Future implementations will support variable cache line lengths and will use these signals.

The following table shows the values of **WBM_SEL_O** signal depending on the Address and the size of the load/store requested.

ADDR_O(1:0)	Size	SEL_O	Data[31:24] valid	Data[23:16] valid	Data[15:8] valid	Data[7:0] valid
xx	Word	1111	Y	Y	Y	y
1x	Half	0011	N	N	Y	Y
0x	Half	1100	Y	Y	N	N
00	Byte	1000	Y	N	N	N
01	Byte	0100	N	Y	N	N
10	Byte	0010	N	N	Y	N
11	Byte	0001	N	N	N	Y

Application Binary Interface.

This section describes the ABI for the MANIK processor. It describes

- Function calling convention. Includes parameter passing and return values
- Layout of data in memory

Function calling convention

The ABI usage of registers are described in the section General Purpose Registers. The first four integral parameters are passed in registers R1 through R4, the rest of the parameters are passed on the stack. Types that are shorter than 32 bits (bytes & chars) are sign or zero extended to 32 bits when passed as parameters. When a structure is passed as

a parameter a copy of it passed to the called function, the copy of the structure may be passed partially in registers are partially on the stack.

Integral return values are all returned in register R1. Return values less the 64 bits in width are returned in the register pair R1-R2. The lower order word is returned in R1. For return values greater than 64 bits in width the caller must provide a pointer to the return area as the first parameter.

Layout of data in memory

MANIK is a BIG Endian machine. The layout of three data types supported shown in the following tables.

Word Data Type (32 bits) (4 bytes)

N	***N+1***	***N+2***	***N+3***
31			***0***
MSB			***LSB***

Half Data Type (16 bits) (2 bytes)

N	***N+1***
15	***0***
MSB	***LSB***

Byte (8 bits)

N	
7	0
Msbit	Lsbit

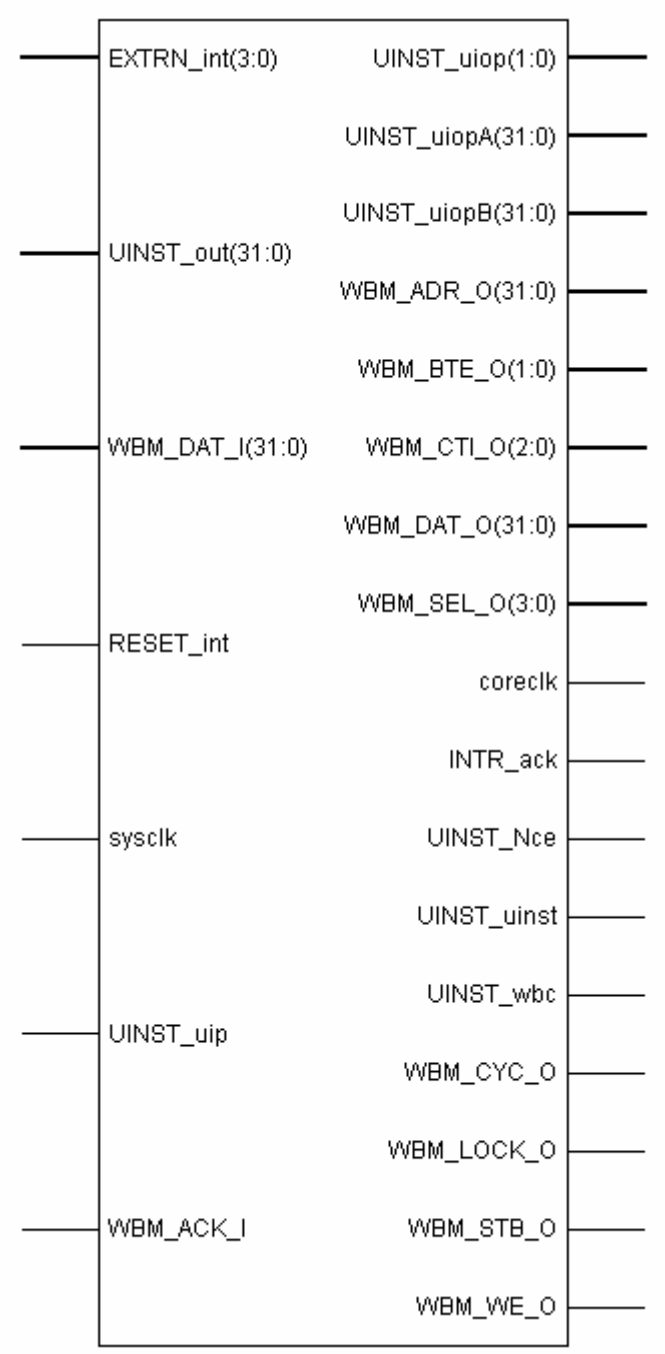
Appendix – A . HDL Instantiation template.

The signal names prefixed with WBM_ are the Wishbone Bus Master interface. See section MANIK Bus Interface. The signals names prefixed with UINST_ are meant for the User Instruction interface. See section USER Instruction interface.

```
component manik2top
  port (
    sysclk      : in  std_logic;
    coreclk     : out std_logic;
    EXTRN_int   : in  std_logic_vector(NUM_INTRS-1 downto 0)   := (others => '0');
    RESET_int   : in  std_logic                                := '0';
    INTR_ack    : out std_logic;
    WBM_DAT_I   : in  std_logic_vector (WIDTH-1 downto 0)      := (others => '0');
    WBM_ACK_I   : in  std_logic                                := '0';
    WBM_ERR_I   : in  std_logic                                := '0';
    WBM_DAT_O   : out std_logic_vector (WIDTH-1 downto 0);
    WBM_SEL_O   : out std_logic_vector (3 downto 0);
    WBM_WE_O    : out std_logic;
    WBM_STB_O   : out std_logic;
    WBM_CYC_O   : out std_logic;
    WBM_LOCK_O  : out std_logic;
    WBM_ADR_O   : out std_logic_vector (ADDR_WIDTH-1 downto 0) := (others => '0');
    WBM_CTI_O   : out std_logic_vector (2 downto 0);
    WBM_BTE_O   : out std_logic_vector (1 downto 0);
    UINST_uiop  : out std_logic_vector(1 downto 0);
    UINST_uinst : out std_logic;
    UINST_Nce   : out std_logic;
    UINST_wbc   : out std_logic;
    UINST_uiopA : out std_logic_vector(UINST_WIDTH-1 downto 0);
    UINST_uiopB : out std_logic_vector(UINST_WIDTH-1 downto 0);
    UINST_uip   : in  std_logic                                := '0';
    UINST_out   : in  std_logic_vector(UINST_WIDTH-1 downto 0) := (others => '0'));
end component;
```

The "**sysclk**" input signal is the primary input clock to the core. The output signal "**coreclk**" is the same as "sysclk" in this implementation.

External interrupts should be connected to the "EXTRN_int" signal, these are level triggered signals; see section External Interrupt for more details.


EXTRN_int(3:0)
UINST_out(31:0)
WBM_DAT_I(31:0)
RESET_int
sysclk
UINST_uip
WBM_ACK_I
UINST_uiop(1:0)
UINST_uiopA(31:0)
UINST_uiopB(31:0)
WBM_ADR_O(31:0)
WBM_BTE_O(1:0)
WBM_CTI_O(2:0)
WBM_DAT_O(31:0)
WBM_SEL_O(3:0)
coreclk
INTR_ack
UINST_Nce
UINST_uinst
UINST_wbc
WBM_CYC_O
WBM_LOCK_O
WBM_STB_O
WBM_WE_O

Appendix – B. Alphabetic list of Instructions.

ADD Add , update Carry

Operation:

```
Rd = Rd + Rb; CY = Carryout
```

Assembler Syntax : add Rd,Rb

Description:
Add the contents of Register D and the contents of Register B, and put the result in Register D.

PSW Flags updated: CY = Carryout;

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

ADDC Add with Carry, update Carry

Operation:

```
Rd = Rd + Rb + CY; CY = Carryout
```

Assembler Syntax: addc Rd,Rb

Description:
Add the contents of Register B, the CY bit (in the PSW), and the contents of Register D; and store the result in Register D.

PSW Flags updated: CY = Carryout.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

ADDF Conditional Add if False

Operation:

```
If (TF=0) then Rd = Rd + Rb;
```

Assembler Syntax: addf Rd,Rb

Description:
Add the contents of Register Rd and Rb, and store the result in Rd; perform the operation only if TF = 0.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

ADDI Add with immediate

Operation:

```
Rd = Rd + sign extended(Simm8)
```

Assembler Syntax: addi Rd,Simm8

Description:

Sign extend 8 bit immediate value to 32 bits and add it to the contents of register Rd, store the result in register Rd. Immediate value can range between –127 to +127.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	S	S	S	S	Rd	Rd	Rd	Rd	S	S	S	S

ADDIF Conditional Add with immediate , if False

Operation:

```
If (TF=0) then
        Rd = Rd + Sign extend (Simm4)
```

Assembler Syntax: addif Rd,Simm4

Description:

Sign extend 4 bit immediate to 32 bits and add to the contents of Register D, store the result into Register D , if the TF flag in the PSW is equal to 0.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	0	Rd	Rd	Rd	Rd	S	S	S	S

ADDIT Conditional Add with immediate, if True

Operation:

```
If (TF=1) then
        Rd = Rd + Sign extend (Simm4)
```

Assembler Syntax: addit Rd,Simm4

Description:
Sign extend 4 bit immediate to 32 bits and add to the contents of Register D, store the result into Register D , if the TF flag in the PSW is equal to 1.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	0	Rd	Rd	Rd	Rd	S	S	S	S

ADDT Conditional Add, if True

Operation:

```
If (TF=1) then
        Rd = Rd + Rb;
```

Assembler Syntax: addt Rd,Rb

Description:
Add the contents of register D, and contents of Register B and put the result into Register D, if the TF flag in the PSW = 1.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

AND Logical Bitwise And

Operation:

```
Rd = Rd & Rb
```

Assembler Syntax: and Rd,Rb

Description:
Logically AND the contents of Register D with the contents of Register B, and store the results in Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

ANDI Logical And with immediate

Operation:

```
Rd = Rd & zero extend(UIMM8);
```

Assembler Syntax: andi Rd,Uimm8

Description:
Zero extend the 8 bit immediate to 32 bits and perform a logical AND with the contents of Register D, the result is store into Register D. The immediate must be a positive integer between 0 and 255.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	U	U	U	U	Rd	Rd	Rd	Rd	U	U	U	U

ASR Arithmetic Shift Right (Dynamic)

Operation:

```
Rd = (signed) Rd >> Rb[5:0]
```

Assembler Syntax: asr Rd,Rb

Description:
Arithmetic shift right the contents of Rd by Rb[5:0] bits; store the result into register D. If Rb[5:0] is greater than 31 then the result is either 0, or –1 depending on the initial value of bit 31 of register D. Bits 31 through 6 of Register B are ignored.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

ASRI Arithmetic with immediate (Static)

Operation:

```
Rd = Rd >> (Uimm4);
```

Assembler Syntax: asri Rd,Uimm4

Description:
Arithmetic shift right the contents of Register D by uimm4 bits, and store the result in Register D. Uimm4 must be a positive integer between 0 and 15.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	Rd	Rd	Rd	Rd	U	U	U	U

CMPEQ Compare for Equal

Operation:

```
if Rd == Rb then
        TF = 1
Else
        TF = 0
```

Assembler Syntax: cmpeq Rd,Rb

Description:
Compare the contents of Register D, with the contents of Register B; if they are equal set the TF in the PSW to 1 else set the flag to 0.

PSW Flags updated: TF flag.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

CMPEQI Compare with Immediate for Equal

Operation:

```
if Rd == sign extend(4 bit immediate) then
        TF = 1
Else
        TF = 0
```

Assembler Syntax: cmpeqi Rd,simm4

Description:
Sign extend 4 bit immediate to 32 bits and compare with the contents of Register D, if they are equal set the TF flag in the PSW to 1 , else set it to 0. The immediate value must be an integer between –8 and +7.

PSW Flags updated: TF flag.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	0	Rd	Rd	Rd	Rd	S	S	S	S

CMPGT Compare for Greater Than (Signed)

Operation:

```
if Rd > Rb (Signed compare) then
        TF = 1
Else
        TF = 0
```

Assembler Syntax: cmpgt Rd,Rb

Description:
Set TF flag in the PSW to 1 if contents of Register D, is greater than contents of Register B else set TF flag to 0 ; the contents of both registers are treated as signed numbers.

PSW Flags updated: TF flag.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

CMPGTI Compare with Immediate for Greater than (signed)

Operation:

```
If Rd > Sign extended (simm4) then
        TF = 1
Else
        TF = 0
```

Assembler Syntax: cmpgti Rd,simm4

Description:
Set TF flag to 1 if the contents of Register D, is greater than sign extended 4 bit immediate value, else set TF flag to 0. The contents of Register D is treated as an unsigned number; the Simm4 must be an integer between –8 and +7.

PSW Flags updated: TF flag.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**Rd**	**Rd**	**Rd**	**Rd**	**S**	**S**	**S**	**S**

CMPHS Compare for Higher or Same (Unsigned)

Operation:

```
If Rd >= Rb then
        TF = 1
Else
        TF = 0
```

Assembler Syntax: cmphs Rd,Rb

Description:
Set TF flag to 1 if the contents of Register D is the equal to or greater than contents of Register B. The contents of both registers are treated as unsigned quantities.

PSW Flags updated: TF Flag.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

CMPLS Compare for Less than or Same (Unsigned)

Operation:

```
If Rd <= Rb then
        TF = 1
Else
        TF = 0
```

Assembler Syntax: cmpls Rd,Rb

Description:
Set TF flag to 1 if the contents of Register D is the equal to or less than contents of Register B. The contents of both registers are treated as unsigned quantities.

PSW Flags updated: TF flag.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

CMPLT Compare for Less Than (Signed)

Operation:

```
If Rd < Rb then
        TF = 1;
Else
        TF = 0;
```

Assembler Syntax: cmplt Rd,Rb

Description:
Set the TF flag to 1 if the contents of Register D is less than the contents to Register B, else set the TF flag to 0. The contents of both registers are treated as signed numbers.

PSW Flags updated: TF flag.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

CMPLTI Compare with Immediate for Less than (Signed)

Operation:

```
If Rd < Sign Extend(simm4) then
        TF = 1;
Else
        TF = 0;
```

Assembler Syntax: ***cmplti*** ***Rd,simm4***

Description:

Set TF flag to 1 if the contents of Registers D, is less than the sign extended 4 bit immediate value. The contents of Register D is treated as a signed number. The 4 bit immediate value must be an integer between –8 and +7.

PSW Flags updated: TF flag.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	0	Rd	Rd	Rd	Rd	S	S	S	S

J Unconditional Branch (PC relative)

Operation:

```
PC = PC + (Sign extended(Simm27) << 1);
```

Assembler Syntax: ***j*** ***[Assembler Label]***

Description:

The PC is updated by adding its contents to a scaled sign extended 27 bit displacement field. The displacement represents the offset to the destination address in half words from the branch instruction. This gives the branch instruction a range of +/- 255 MB.

PSW Flags updated: None.

Instruction Format:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	1	0	S	S	S	S	S	S	S	S	S	S	S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S

JF Conditional Branch if False

Operation:

```
If TF = 0 then
        PC = PC + (sign extend (simm27)) << 1;
Else
        PC = PC + 4;
```

Assembler Syntax: ***jt*** ***<Assembler Label>***

Description:
If the TF flag in the PSW is zero then the PC is updated by adding its contents to scaled sign extended 27 bit displacement field; otherwise the PC is incremented by 4. The displacement represents the offset to the destination address in half words from the branch instruction. This gives the branch instruction a range of +/- 255 MB.

PSW Flags updated: None.
Instruction Format:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	0	1	S	S	S	S	S	S	S	S	S	S	S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S

JL Branch to subroutine; Update RA

Operation:

```
RA = PC + 4;
PC = PC + ((sign extend (simm27)) << 1;
```

Assembler Syntax: jl <Assembler Label>

Description:
Jump and Link ; The return address (PC+4) is saved in the Special Function Register 1 (RA). The PC is updated by adding its contents to scaled sign extended 27 bit displacement field. The displacement represents the offset to the destination address in half words from the branch instruction. This gives the branch instruction a range of +/- 255 MB.

PSW Flags updated: None.

Instruction Format:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	1	1	S	S	S	S	S	S	S	S	S	S	S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S

JR Branch Register Indirect

Operation:

```
PC = (Rb) & 0xFFFFFFFE;
```

Assembler Syntax: jr Rb

Description:
The PC is updated with the contents of Register B. The lowest order bit of Register B is ignored.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	0	0	0	0	Rb	Rb	Rb	Rb

JRL Branch to Subroutine; Register Indirect; Update RA

Operation:

```
RA = PC + 4;
PC  = (Rb ) & 0xFFFFFFFE;
```

Assembler Syntax: jrl Rb

Description:

The return address address (PC+2) is saved in the Special Function Register 1 (RA). The PC is updated with the contents of Register B, the lowest order bit of the register is ignored. The assembler inserts a NOP instruction after all "**jrl**" instruction to compensate for the size difference between the size of a "**jl**" and "**jrl**" instruction.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	0	0	0	0	Rb	Rb	Rb	Rb

JSFR Branch Special Function Register Indirect

Operation:

```
PC = (SFRn) & 0xFFFFFFFE;
If SFRn == SFR2 (IPC) then
        IP_flag = 0 ;
```

Assembler Syntax: jsfr SFRn

Description:

The PC is updated with the contents of Special Function Register (n); the least significant bit of the SFRn is ignored. This instruction is typically used to return from leaf functions; or from return from interrupt routines. If the SFR number is 2 (i.e. Interrupt PC) then the ip_flag in the PSW is also cleared.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	0	0	0	0	0	Sfrn	Sfrn

JT Conditional Branch if True

Operation:

```
If TF = 1 then
       PC = PC + (sign extend (simm27)) << 1;
Else
       PC = PC + 4;
```

Assembler Syntax: ***jt*** ***<Assembler Label>***

Description:
If the TF flag in the PSW is set then the PC is updated by adding its contents to scaled sign extended 27 bit displacement field; otherwise the PC is incremented by 4. The displacement represents the offset to the destination address in half words from the branch instruction. This gives the branch instruction a range of +/- 255 MB.

PSW Flags updated: None.

Instruction Format:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	0	1	S	S	S	S	S	S	S	S	S	S	S

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S

LDR[BH] Load Register from memory

Operation:

```
Rd = MEM[Rb + unsigned IMM4 << {0,1,2}]
```

Assembler Syntax:

ldr Rd,Uimm4(Rb)
ldrb Rd,Uimm4(Rb)
ldrh Rd,Uimm4(Rb)

Description:
The effective address of the load operation is computed by adding the scaled unsigned immediate value to the contents of Register B. The scaling is performed by left shifting the unsigned immediate 4 bit value by the size of the load operation. The effective address is aligned to the size boundary of the load operation (i.e. for word loads the lowest two bits are forced to zero; for half word loads the lowest order bit is forced to zero). For load sizes less than word the Destination register (Rd) is zero extended.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Sz	Sz	0	U	U	U	U	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

```
Sz =  00 -> Word      Unsigned IMM4 << 2;
      01 -> Half Word Unsigned IMM4 << 1;
```

```
10 -> Byte      Unsigned IMM4;
```

LDRPC Load Word(32bits) from literal pool (PC relative)

Operation:

```
Rd = MEM[(PC + UIMM8 << 2) & 0xFFFFFFFC]
```

Assembler Syntax: *ldrpc Rd,<assembler Label>*
ldrpc Rd,<Immediate value>

Description:
The assembler creates a literal table entry containing the address of the label or the immediate value and sets the eight bit immediate value to point to the table entry. The effective address is computed by left shifting the 8 bit immediate by two and zero extending the result, this is added to the contents of PC and the lower order two bits are forced to zero.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	U	U	U	U	Rd	Rd	Rd	Rd	U	U	U	U

LSL Logical Shift Left (Dynamic)

Operation:

```
Rd = Rd << Rb[5:0];
```

Assembler Syntax: *lsl Rd,Rb*

Description:
The contents of the Register D, is left shifted by Rb[5:0] bits, and the results are stored in Register B; if Rb[5:0] > 31 then Rd = 0.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

LSLI Logical Shift Left with immediate (Static)

Operation:

```
Rd = Rd << UIMM4 ;
```

Assembler Syntax: *lsli Rd, Uimm4*

Description:
The contents of Register D is left shifted by Uimm4 bits and the results are stored back into Register D. The Uimm4 value must be an integer between 0 and 15.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	Rd	Rd	Rd	Rd	U	U	U	U

LSR Logical Shift Right (Dynamic)

Operation:

```
Rd = Rd >> Rb[5:0];
```

Assembler Syntax: ***lsr Rd,Rb***

Description:
The contents of Register D is right shifted by Rb[5:0] bits; the result is stored back into Register D. If Rb[5:0] is greater than 31 then Rd = 0;

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

LSRI Logical Shift Right with immediate (Static)

Operation:

```
Rd = Rd >> UIMM4;
```

Assembler Syntax: ***lsri Rd, Uimm4***

Description:
The contents of Register D is right shifted by Uimm4 bits and the results are stored back into Register D. The Uimm4 value must be an integer between 0 and 15.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	Rd	Rd	Rd	Rd	U	U	U	U

MFSFR Move From SFR to GPR

Operation:

```
Rd = SFRn;
```

Assembler Syntax: ***mfsfr Rd,SFRn***

Description:
The contents of the Special Function Register n is stored into the Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	Rd	Rd	Rd	Rd	0	0	SFR	SFR

```
SFR    00     SFR0   PSW
       01     SFR1   RA
       10     SFR2   IPC
       11     SFR3   TIMER
```

MOV Unconditional move from GPR to GPR

Operation:

```
Rd = Rb;
```

Assembler Syntax: ***mov Rd, Rb***

Description:
The contents of Register B is stored into Register D.
PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

MOVF Conditional Move from GPR to GPR, if False

Operation:

```
If TF==0 then
        Rd = Rb;
```

Assembler Syntax: ***movf Rd, Rb***

Description:
The contents of Register B are conditionally moved to Register D if TF flag is 0.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

MOVI Unconditional Move immediate to GPR

Operation:

```
Rd = sign extend(SIMM8);
```

Assembler Syntax: ***movi Rd, Simm8***

Description:
The eight immediate value is sign extended and stored into Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	S	S	S	S	Rd	Rd	Rd	Rd	S	S	S	S

MOVT Conditional Move from GPR to GPR, if True

Operation:

```
If TF == 1 then
        Rd = Rb;
```

Assembler Syntax: ***movt Rd, Rb***

Description:
The contents of Register B are conditionally moved to Register D if TF flag is 1.
PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

MTSFR Move from SFR to GPR

Operation:

```
SFRn = Rb;
```

Assembler Syntax: ***mtsfr SFRn, Rb***

Description:
The value of SFRn is updated with the contents of Register B. Currently only PSW (SFR0), IPC (SFR2) and TIMER (SFR3) can be updated by software.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	0	0	Sfr	Sfr	Rb	Rb	Rb	Rb

MULT Multiply

Operation:

```
Rd = Rd * Rb;
```

Assembler Syntax: ***mult Rd,Rb***

Description:
The contents of Register D and the contents of Register B are multiplied and the lower order 32 bits are stored in the Register D. The results are the same regardless of wether the source operands are considered signed or unsigned.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

MULTI Multiply with immediate

Operation:

```
Rd = Rb * Uimm4;
```

Assembler Syntax: ***multi Rd, Uimm4***

Description:
The contents of Register D is multiplied by the zero extended unsigned 4 bit immediate value and the results are stored in back in Register D. The Uimm4 must be an integer in between 0 and 15.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	Rd	Rd	Rd	Rd	U	U	U	U

OR Logical OR

Operation:

```
Rd = Rd | Rb;
```

Assembler Syntax: ***or Rd, Rb***

Description:
A bitwise logical “or” operation is performed with the contents of Register D & the contents of Register B; the results are stored back into Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

STR[BH] Store Word(32 bits) to memory

Operation:

```
MEM[Rb + (UIMM4) << {0,1,2}] = Rd;
```

Assembler Syntax: ***str Rd, Uimm4(Rb)***
strh Rd, Uimm4(Rb)
strb Rd, Uimm4(Rb)

Description:

Store the contents of Register D into memory. The store operation can be performed in three sizes Word, Half word (h), or Byte (b). The effective address is computed by scaling the Uimm4 value (left shift by 2 for word, left shift by 1 for half word) and adding the result to the contents of Register B. The effective address is aligned to the size boundary of the store operation (i.e. for word loads the lowest two bits are forced to zero; for half word loads the lowest order bit is forced to zero).

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Sz	Sz	1	U	U	U	U	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

```
SZ -> 10 Byte
      00 Word      UIMM4 << 2
      01 Half word UIMM4 << 1
```

SUB Subtract; Update carry

Operation:

```
Rd = Rd - Rb;
CY = carry out from the subtract;
```

Assembler Syntax: ***sub Rd, Rb***

Description:
The contents of Register B is subtracted from the contents of Register D and the result stored in Register D. The CY flag is updated with the carry out from this operation.

PSW Flags updated: CY flag carryout from the subtract.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

SUBC Subtract with carry; update carry

Operation:

```
Rd = Rd - Rb - C;
CY = Carry out from the subtract
```

Assembler Syntax: ***subc Rd, Rb***

Description:
The contents of Register B and the CY bit is subtracted from the contents of Register D and the result stored in Register D. The CY flag is updated with the carry out from this operation.

PSW Flags updated: CY flag carryout from the subtract.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

SUBF Conditional Subtract; if false

Operation:

```
If TF == 0 then
        Rd = Rd - Rb;
```

Assembler Syntax: ***subf Rd, Rb***

Description:
If the TF flag is zero subtract the contents of Register B from the contents of Register B and store the result in Register D. The operation is not performed if the TF is set.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

SUBT Conditional Subtract; if true

Operation:

```
If TF == 1 then
        Rd = Rd - Rb;
```

Assembler Syntax: ***subt Rd, Rb***

Description:
If the TF is set then subtract the contents of Register B from the contents of Register B and store the result in Register D. The operation is not performed if the TF is set.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

SWINT Software Interrupt

Operation:

```
IPC = PC;
PC = 0x8;
PSW[23:16] = SW (swint code from instruction field);
IP & SW flag = 1;
```

Assembler Syntax: swint [0-255]

Description:

Software interrupt; the address of the instruction following the swint instruction is captured in the IPC (SFR) ; IP & SW flags in the PSW are set to 1. The lowest order 8 bits from the instruction field are copied into the PSW bits [23:16].

PSW Flags updated: SW, IP flags & SWINT.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	Sw	Sw	Sw	Sw	Sw	Sw	Sw	Sw

SXB Sign extend byte

Operation:

```
Rd = sign extend(Rb[7:0])
```

Assembler Syntax: sxtb Rd, Rb

Description:

Sign extend the lower order byte (eight bits) , and put the result in Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	0	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

SXH Sign Extend half word

Operation:

```
Rd = sign extend (Rb[15:0])
```

Assembler Syntax: ***sxh*** ***Rd, Rb***

Description:
Sign extend the lower order half word of Register B (16 bits) and put the result in Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	1	0	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

UDI[0-3] User Defined instructions

Operation:

```
Rd = result of user defined operation(Rd, Rb)
```

Assembler Syntax: ***udi[0,1,2,3]*** ***Rd, Rb***

Description:
For more details see section USER Instruction interface.
PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	FN	FN	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

XHW Exchange half word

Operation:

```
Rd = (Rb << 16) | ((Rb >> 16) & 0x0000FFFF);
```

Assembler Syntax: ***xhw*** ***Rd, Rb***

Description:
The Upper half word Register B is placed in the Lower half word of Register D; the lower half word of Register B is placed in the Upper half word of Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	0	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

XOR Logical XOR

Operation:

```
Rd = Rd ^ Rb;
```

Assembler Syntax: ***xor*** ***Rd, Rb***

Description:
The contents of Register B is bitwise exclusive or ed with the contents of Register D and the results put into Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb

ZXH Zero Extend

Operation:

```
Rd = zero extend (Rb[15:0])
```

Assembler Syntax: ***zxh*** ***Rd, Rb***

Description:
Zero extend the lower half word of Register B and put the result in Register D.

PSW Flags updated: None.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	1	1	Rd	Rd	Rd	Rd	Rb	Rb	Rb	Rb