WUT-4 ISA and System Architecture

This is the fourth computer design from Wholly Unnecessary Technology (WUT). Hence, the WUT-4. With this design, WUT has moved on from the CISC design of YARC to a RISC (ish) instruction set. The WUT-4 has 64kib of code address space and 64kib of data address space for each of several user processes. It has 16-bit little endian words and supports both byte and word data addressing. Its system architecture includes a privileged mode with its own 64+64kB code and data, precise exceptions, and a simple memory management unit (MMU) that supports paging.

The WUT-4 architecture is based on the Ridiculously Simple Computer (RiSC) designed by Prof. Bruce Jacob, but the design has evolved significantly since conception (and it is no longer ridiculously simple!). Changes include the addition of byte addressing, split code and data spaces, CPU flags and conditional branches with a 10-bit offset, special registers, a memory management unit, and a privileged mode. The original WUT-4 architecture document, which included only some of these changes, has been preserved.

Instructions

Opcode high order bits 0x0 through 0xE (binary 000… though 1110…) are called "base" instructions. All base instructions have embedded immediate values. The immediate value of the opcode 1110…, called "JAL" in the table below, must be positive, producing 0xE in the high-order 4 bits. In the assembler, the third (immediate) operand is optional for all base instructions and defaults to 0 if not given.

The opcodes 1111… (0xF…) are the "extended" codes. These are broken down into successive sets of 7 3-operand, 7 2-operand, 7 single-operand, and 8 0-operand instructions called XOPs, YOPs, ZOPs, and VOPs respectively (28 instructions total plus 0xFFFF, which generates an illegal instruction trap, as does the opcode 0 as a special case). There are 8 16-bit general purpose registers, with register 0 usually reading as 0 and writing as a "black hole" (some instructions implement an alternative definition for register 0). There are also special registers described below.

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDW	0	0	0	i6	–	–	–	–	–	i0	rB	rB	rB	rA	rA	rA
LDB	0	0	1	i6	–	–	–	–	–	i0	rB	rB	rB	rA	rA	rA
STW	0	1	0	i6	–	–	–	–	–	i0	rB	rB	rB	rA	rA	rA
STB	0	1	1	i6	–	–	–	–	–	i0	rB	rB	rB	rA	rA	rA
ADI	1	0	0	i6	–	–	–	–	–	i0	rB	rB	rB	rA	rA	rA

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDW	0	0	0	i6	–	–	–	–	–	i0	rB	rB	rB	rA	rA	rA
LUI	1	0	1	i9	–	–	–	–	–	–	–	–	i0	rA	rA	rA
BRx	1	1	0	i9	–	–	–	–	–	–	–	–	i0	b2	b1	b0
JAL	1	1	1	0	i5	–	–	–	–	i0	rB	rB	rB	rA	rA	rA
XOP	1	1	1	1	x2	x1	x0	rC	rC	rC	rB	rB	rB	rA	rA	rA
YOP	1	1	1	1	1	1	1	y2	y1	y0	rB	rB	rB	rA	rA	rA
ZOP	1	1	1	1	1	1	1	1	1	1	z2	z1	z0	rA	rA	rA
VOP	1	1	1	1	1	1	1	1	1	1	1	1	1	v2	v1	v0

The rA, rB, and rC fields each encode 1 of the 8 general registers or, in the case of branches and vops, opcde values. The rA field is the source or destination of memory operations and the target of register to register operations. iN:i0 are N-bit immediate values. The x's, y's, z's, and v's are opcode bits, as are b2:b0 for the BRx instruction. XOP, YOP, ZOP, and VOP each break down into 7 distinct instructions, each with 3, 2, 1, or 0 register arguments. These are detailed in tables that follow.

There are special registers, including a link register LINK and a flags register F containing the arithmetic flags. The numbering of special registers is found in a later section. Kernel mode provides read/write access to usermode registers as described below.

The assembly language is written in destination [, source [, source]] ("DSS") order.

LDW, LDB : load word or byte rA = mem[rB + imm7]. When a byte is loaded, it is sign extended into the upper byte.

STW, STB - store word or byte mem[rB + imm7] = rA.

ADI : rA = rB + imm7.

If rB is 0, the source value is all zeroes. If rA is 0, the result is stored in the link register LINK. The condition codes are updated. This is the only base instruction that sets the condition codes. This instruction is often used as part of a pattern with a preceding Load Upper Immediate (LUI) instruction to load a 16-bit immediate value.

BRx : if the condition specified by b2:b0 is true, branch to PC + 2 + imm10.
        The rA field acts as opcode (really "condition selector") bits, called b2:b0:

if the b2:b0 (ra field) value is…	then the opcode is…	and the branch condition is…
0	br	true (unconditional branch)
1	brl	true (unconditional branch to subroutine; **writes link register**)
2	brz or breq	eq / z
3	brnzn or brneq	neq / nz
4	brc or bruge	uge / c
5	brnc or brult	ult / nc
6	brsge	sge (s flag == v flag; s XNOR v)
7	brslt	slt (s flag != v flag; s XOR v)

LUI : Load upper immediate: rA = imm10 << 6. Does not set condition codes. If rA is 0, the link register is loaded.

JAL: PC = rB15:6 + imm5:0. The 16 bit destination address is constructed by taking the high-order 10 bits from the source register rB and appending the low order 6 bits from the immediate field in the instruction.The resulting code address must be even. If rB is 0, it refers to the link register. rA = old PC + 2. If rA is 0, it also refers to the link register. Note that this means there is no JMP instruction; the return address is always written to an actual register. The register used to construct the 16-bit address can be reused as a target. There is an unconditional relative branch(BR) instruction that does not require use of a register and an indirect jump (JI)

XOP: 1 of 7 operations encoded in the x2:x0 fields when bits 15:12 are 1.
YOP: 1 of 7 operations encoded in the y2:y0 field when bits 15:9 are 1.
ZOP: 1 of 7 operations encoded in the z2:z0 field when bits 15:6 are 1.
VOP: 1 of 7 operations encoded in the v2:v0 field when bits 15:3 are 1.

XOPs (below) are 3-operand ALU instructions. Subtract is rA ⇐ rB - rC. SBB is subtract with borrow. Unfortunately there is no opcode space for a reverse subtract; it can be accomplished via NEG rB; rA ⇐ rB + rC. These instructions affect the CPU flags. These instructions **never** read or update the LINK special register; a zero register specifier produces the value 0 and a 0 target specifier discards the result of the operation.

SBB	1	1	1	1	0	0	0	rC	rC	rC	rB	rB	rB	rA	rA	rA
ADC	1	1	1	1	0	0	1	rC	rC	rC	rB	rB	rB	rA	rA	rA
SUB	1	1	1	1	0	1	0	rC	rC	rC	rB	rB	rB	rA	rA	rA
ADD	1	1	1	1	0	1	1	rC	rC	rC	rB	rB	rB	rA	rA	rA
XOR	1	1	1	1	1	0	0	rC	rC	rC	rB	rB	rB	rA	rA	rA
OR	1	1	1	1	1	0	1	rC	rC	rC	rB	rB	rB	rA	rA	rA
AND	1	1	1	1	1	1	0	rC	rC	rC	rB	rB	rB	rA	rA	rA
YOP	1	1	1	1	1	1	1	y2	y1	y0	rB	rB	rB	rA	rA	rA

YOP's (below) are 2-operand operations. LSP (load special) loads the rA with a value from the special register addressed by rB. SSP stores the value in rA to the special register addressed by rB. LSI loads the special register addressed by rB into the memory location addressed by rA. Note that this is a 16 bit transfer so the address must be aligned.. To store a single byte from a special register use LSP and STB. SSI stores to the special register addressed by rA from the memory location addressed by rB. LCW loads rA from the code space using the address in rB, page permissions allowing.The load is always 16 bits and must be aligned. There is no "store code word" instruction.. TST compares the two register operands by subtracting (rA - rB), setting the flags but discarding the result.

YOP 5 = SYS:
When rB is 0, the rA field selects one of 8 system call entrypoints. The WUT-4 transitions to kernel mode for the next fetch. The return PC is stored in the kernel mode interrupt return special register, IRR. The 8 legal instruction codes are called SYS 0 through SYS 7 in the assembler.

An illegal instruction trap occurs if SYS is executed and rB is not 0

LSP	1	1	1	1	1	1	1	0	0	0	rB	rB	rB	rA	rA	rA
LSI	1	1	1	1	1	1	1	0	0	1	rB	rB	rB	rA	rA	rA
SSP	1	1	1	1	1	1	1	0	1	0	rB	rB	rB	rA	rA	rA
SSI	1	1	1	1	1	1	1	0	1	1	rB	rB	rB	rA	rA	rA
LCW	1	1	1	1	1	1	1	1	0	0	rB	rB	rB	rA	rA	rA
SYS	1	1	1	1	1	1	1	1	0	1	rB	rB	rB	rA	rA	rA
TST	1	1	1	1	1	1	1	1	1	0	rB	rB	rB	rA	rA	rA

LSP	1	1	1	1	1	1	1	0	0	0	rB	rB	rB	rA	rA	rA
LSI	1	1	1	1	1	1	1	0	0	1	rB	rB	rB	rA	rA	rA
SSP	1	1	1	1	1	1	1	0	1	0	rB	rB	rB	rA	rA	rA
SSI	1	1	1	1	1	1	1	0	1	1	rB	rB	rB	rA	rA	rA
LCW	1	1	1	1	1	1	1	1	0	0	rB	rB	rB	rA	rA	rA
SYS	1	1	1	1	1	1	1	1	0	1	rB	rB	rB	rA	rA	rA
ZOP	1	1	1	1	1	1	1	1	1	1	z2	z1	z0	rA	rA	rA

Most ZOP's (below) are single operand arithmetic ops. But JI (jump Indirect) loads all 16 bits of the PC from the specified register. If the register field is zero (0b000), it refers to the link register. JI is used for computed jumps and also to implement the return instruction alias described later. SXT sign extends the lower byte into the upper byte. DUB copies the upper byte of rA to the lower. SXT clears the carry flag and sets the zero flag based on the 16-bit result. SRA and SRL shift the register arithmetically right (sign fill) or logically right (zero fill). Left shifts are supported in the assembler as aliases; they generate an ADD or ADC opcode and set the flags as would be expected from the adds. SRA and SRL set the carry flag to the value of the bit shifted out.

NOT	1	1	1	1	1	1	1	1	1	1	0	0	0	rA	rA	rA
NEG	1	1	1	1	1	1	1	1	1	1	0	0	1	rA	rA	rA
DUB	1	1	1	1	1	1	1	1	1	1	0	1	0	rA	rA	rA
SXT	1	1	1	1	1	1	1	1	1	1	0	1	1	rA	rA	rA
SRA	1	1	1	1	1	1	1	1	1	1	1	0	0	rA	rA	rA
SRL	1	1	1	1	1	1	1	1	1	1	1	0	1	rA	rA	rA
JI	1	1	1	1	1	1	1	1	1	1	1	1	0	rA	rA	rA
VOP	1	1	1	1	1	1	1	1	1	1	1	1	1	v2	v1	v0

And last but not least VOPs. CCF and SCF clear and set the carry flag. DI and EI disable and re-enable external interrupts, respectively. HLT is a real instruction, but BRK may be implemented only in the emulator. RTI, return from interrupt, returns to the address held in the IRR, transitions to user mode, and enables interrupts. DI, EI, HLT, and RTI generate an illegal instruction fault if executed in usermode. DIE always generates an illegal instruction fault.

CCF	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
SCF	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
DI	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
EI	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
HLT	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
BRK	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
RTI	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
DIE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Assembler Instruction Details

LDW r0, r0+0 (which would assemble to 0x0000) generates an assembler error. The opcode 0x0000, if executed, generates an illegal instruction trap. This is a special case. The DIE mnemonic generates 0xFFFF. It should be used for aborts.

Alias (or pseudo-) instructions supported by the assembler that expand to other instructions include:

ldi rT, imm16 (load immediate with a value in 0..0xFFFF) is a nontrivial builtin assembler alias:

If imm16 < 0x40, it expands to ADI rT, r0, imm6 ("add short immediate", ADI) which moves the small value to the target register.
if imm16 & 0xFFC0 == imm16, it expands to LUI rT, imm10 ("load upper immediate", LUI)
Otherwise, it expands to (LUI rT, (imm16&0xFFC0)>>6 ; ADI rT, r0, imm16&0x3F) ("load full size immediate").
If rT is 0, the LINK register is the target.

jal label
Shorthand for jal link (0), label

jal rT, label
Expands to lui rT, (label&0xFFC0)>>6 ; jal rT, rT, (label&0x3F).
rT may be LINK(0).

jal rT, rS, label
Expands to lui rT, (label&0xFFC0)>>6 ; jal rT, rS, (label&0x3F).
rT, rS, or both may be LINK.
The rA field holds rT (the PC save register). The rB field holds rS.

This fully general form subsumes all the shorthands.

mv   rT, rS
       Expands to adi rT, rS, 0
       moves (assigns) the value in rS to RT

srr rA, rB, imm7 (LI rB, imm7 ; LSP rA, rB) = li ; srr - "Special Register Read"
       Load rA from one of the 128 special registers at imm7 using rB.
        Special register space is word addressable.

srw rA, rB, imm7 (LI rB, imm7 ; SSP rA, rB) = li ; srw - "Special Register Write"
       Store rA to one of the 128 special registers at imm7 using rB.
        Special register space is word addressable.

ret [rN] - return
       Implemented as JI rN. Rn may be LINK (0).
       If no argument is given, rN is LINK. rN is modified.

sla rN, sll rN - shift left arithmetic / logical
       SLA is implemented as ADC rN, rN, rN; SLL is implemented as ADD rN, rN, rN. Unlike
SRA and SRL,
       these two instructions do not set the carry flag to the "lost" bit–rather they set it like an
add (since they are implemented as adds.).

SYS instruction details: the WUT-4 enters kernel mode before fetching the target of the SYS
instruction, so SYS accesses one of the vectors in low kernel code space. The vectors are each
2 words, allowing them to hold LUI/JAL pairs that can reach all of kernel code space. By
convention, kernel general register 5 is reserved for assembling the jump address.

LSP, LSI, SSP, and SSI (found in the table of YOPs, above) exist as mnemonics. The
two-instruction SRR and SRW forms exist for convenience because in most situations, the 7-bit
special register address is not already sitting in a register.

Memory consistency

The WUT-4 is sequentially consistent. Memory operations are never hoisted, deferred,
reordered or speculated. Physical accesses must appear in program order when viewed by
external agents (I/O devices or even test equipment).

Instructions that reference memory either fault before issuing the memory reference or do not
fault. The effect is that memory references are never retried; they occur once per successful
retirement of the referencing instruction. Register sources of memory writes may be read twice if
the memory write results in a fault (e.g. page fault). For general registers and most special

registers this is harmless, but care must be taken when reading the free-running cycle counter (SPR 6 and 7): the SSI (store special register indirect) instruction should not be used to read the counter. Instead, use the standard algorithm to ensure the upper and lower halves are consistent, storing the value in two general registers, and then store the consistent result in memory.

Context

This document has referred to "user space" as if there was only one. In fact, the architecture guarantees the existence of a minimum of 255 user spaces 1..0xFF in addition to the single kernel space 0. At any point in time, exactly one user space is potentially active. This user space is identified by the Context register. Kernel space is always potentially active. A properly designed kernel may maintain up to 255 user processes and switch between them simply by writing the context register (no per-process state need be saved to data memory when switching between contexts). The context register may be read and written as a kernel-mode SPR.

When the WUT-4 is in privileged mode, the kernel context (0) is implicit. Reading the context register returns the most recent user context. The value in the context register defines the user mode register resources that the kernel can modify. The kernel may write the context register as described in the next section, with immediate effect. When the kernel executes a RTI that enters user mode, the hardware will attempt to execute in the context identified by the Context register. This description implies that the kernel should set the context register to a nonzero value, typically 1, before constructing the initial usermode context and executing RTI to set it running.

Special Registers (SPRs)

Special register space is 128 words, word-addressed. Only special registers 0..7 may be accessed when executing in usermode. Usermode accesses to SRs 8..127 generate an illegal instruction fault. The link register and CPU flags are per-context. Other special registers are singletons.

The SPRs are defined as follows:

SPR	Name	Details
0	LINK	Accessed in place of "black hole" (0x000) by certain instructions. Reads and writes are effective.
1	FLAGS	Arithmetic flags C (least significant bit), Z, N, V. Bits 4..15 undefined. Reads and writes are effective.
2..5	(undefined)	Reads are undefined. Writes are ignored.
6	CYCLO	Machine cycle counter, low 16 bits. Writes are ignored.
7	CYCHI	Machine cycle counter, high 16 bits. Writes are ignored.

8	IRR	Interrupt return register. Reads and writes are effective.
9	ICR	Interrupt cause register. Writes are ignored. Contains fault cause, e.g. page fault, etc.
10	IDR	Interrupt data register. Writes are ignored. Contains fault address.
11	ISR	Interrupt state register. Reads and writes are effective. Contains previous mode.
12..14	(reserved)	Reserved. Reads are undefined. Writes are ignored.
15	CONTEXT	User mode context register. Reads and writes are effective.
16..23	USERGEN	User mode general registers from previous context. Reads and writes are effective.
24..31	(reserved)	Reserved. Reads are undefined. Writes are ignored.
32..47	USER CODEMMU	User context code pages. Reads and writes are effective.
48..63	USER DATAMMU	User context data pages. Reads and writes are effective.
64..79	KERN CODEMMU	Kernel code pages. Reads and writes are effective.
80..95	KERN DATAMMU	Kernel data pages. Reads and writes are effective.
96..127	I/O REGISTERS	32 I/O registers. See description below.

LINK accesses the dedicated link register. The CPU flags can be read or written in the low order 4 bits of FLAGS (C in bit 0, Z, N, V in bit 3). The flags register contains additional bits visible only in kernel mode: Trap (T) in bit 8, if set, causes the next user instruction executed in user mode to trap. Interrupt Enable (IE) bit 9 reflects the state of the interrupt enable flag. Writing to bit 8 is effective, while bit 9 is read only (the bit is affected by the DI and EI instructions, by the occurrence of interrupts, and by the RTI instruction).

CYCLO and CYCHI provide a machine cycle counter that is reset at power on. The counter will not overflow for at least 7 * 24 hours of 3600 seconds each + 1 second. To read the counter, code should read the high word, then the low word, then the high word again. If the high words differ, the process should be repeated. As noted in the section on memory consistency, the SSI (store special register indirect) instruction should not be used to read the counter (a page fault on the store will cause a double read of the counter and the second read of the counter may differ significantly from the first).

MMU

The kernel special register space contains the direct-mapped MMU. There are 16 MMU registers for code and an additional 16 MMU registers for data in the currently executing context. The MMU supports 64k bytes of process virtual code space and 64k bytes of process data space. Since the MMU does not expand the virtual address space, direct mapping is sufficient: the MMU is not a TLB.

There is one set of 32 registers for each context including context 0 (kernel). The kernel accesses MMU entries at special register addresses as shown in the table above. Only the kernel's own and the MMU for the currently selected user context are accessible. The kernel should not change the code MMU entry addressing the currently-executing page. The hardware provides a minimum of 256 complete sets of MMU registers, just as it does for general registers.

The MMU is always enabled. Hardware initialization places the WUT-4 in privileged mode and loads 0 into kernel code MMU slot 0 and data MMU slot 0 before starting execution at code at address 0. This provides a single 4096 byte shared page for the code and data of the first-level bootstrap. Hardware may provide a ROM (possibly a shadow ROM) at this address. Emulation may load a binary into emulated program RAM as part of initialization.

Each 16-bit MMU register is split into 12 bits that become bits 12:23 of the physical address and 4 control bits, allowing $2^{12} = 4096$ physical pages each having 2^{12} addresses (16MiB physical address space). The four control bits are defined as RRPP where RR is reserved and PP is the permission. The PP bits are encoded so that 00 (the initialization state of code and data context 0 MMU slot 0) is all permissions (RWX) and 11 is invalid (any access causes a page fault exception). For code space, 01 is execute-only; for data space, 01 is read-only. Execute-only code pages are readable to the process, e.g. using the LCW instruction. The value 10 is reserved.

IO Space

The last 32 words of special register space 96..127 form a small I/O address space. There is a console UART at addresses 96 through 99. Bytes may be read in the LS bits of address 96 and written to the LS bits of address 97. Addresses 98 and 99 are reserved for control registers. In the functional simulator, there are no control registers. Bytes written to address 96 are routed to the simulator standard output; reads from address 97 pull bytes from the simulator standard input.

The WUT-4 may include a scheme for priority interrupts. This would include a way of detecting the presence of the interrupt controller and definition of its registers, which would appear in special register space. Addresses 124..127 are reserved for this TBD design.

Memory-mapped I/O is of course supported.

Interrupts, Traps, Faults, and Exceptions

The WUT-4 supports both interrupts and traps/faults/exceptions: the latter three words are used interchangeably. I refer to these as *interrupts* (excluding TFEs), *TFEs* (excluding interrupts), or *ITFEs* (inclusive) below. This vocabulary helps avoid word games over the definitions of "fault", "exception", and "trap".

Traps, faults, and exceptions (TFEs) occur in the course of instruction execution and cannot be deferred because they prevent the current instruction from completing normally. Interrupts have external causes, occur asynchronously, and can be deferred ("disabled"). Execution of the SYS instruction also causes a TFE. Processing is similar in all cases.

The WUT-4 includes a privileged mode that gives access to several resources: kernel code and data memory, certain special registers, and a complete set of kernel-mode general registers. When an ITFE occurs, the WUT-4 enters privileged mode. The next fetch is made from one of several locations in low *kernel* memory called "vectors". The PC of the first instruction not completed is stored in a privileged mode special register called the interrupt return register (IRR). The previous mode (privileged or user) is also stored, along with the cause of the exception, in a privileged mode special register called the interrupt cause register (ICR). Additional information, e.g. the fault address for a page fault, may be stored in the interrupt data register (IDR).

The instruction at the vector must be a flow control instruction and should not alter the link register. In practice it is an LUI r5,target;JAL pair (kernel general register 5 is reserved for this purpose). It could be HLT or the instruction pair BRK; RTI which invokes debugging services from the emulator.

Since ITFEs are disabled when they occur (at entry to privileged mode), additional interrupts will not be serviced. If a TFE condition occurs in privileged mode with ITFEs disabled (a "double fault"), the W4 halts. (For "reliable" operation, the WUT-4 can be configured, in hardware, to reset instead of halting.) Execution of a DIE instruction at the vector will result in a double fault.

The kernel can now proceed to service the ITFE by updating device registers, transferring data, changing the virtual memory mapping, killing a process, etc. When such processing is complete, the kernel can execute an RTI (return from trap or interrupt) instruction. This changes the machine state to the previous state, transfers the contents of the IRR to the PC, and re-enables ITFEs.

A simple kernel implementation would prohibit any nesting of ITFEs. The entire handler would run with interrupts disabled and would ensure that no TFEs occur. A more complex kernel might choose to save its state on a kernel stack and re-enable interrupts without returning from the interrupt using the EI instruction. This approach allows kernel code to encounter and handle nested ITFEs that occur in privileged mode. Each nested ITFE must be matched by an instruction sequence that reloads ("pops") the IRR and ICR from the stack and then executes RTI.

Unresolved issues, additional features, etc.

TODO

1. Need a way for the kernel to read and write the memory of the most recently active user context (for copyin and copyout). **No answer besides map the page. Additional information: even if an instruction is added to perform kernel mode memory access through the most recent user MMU, the kernel would have to worry about page faults causing double faults. This is not much easier than just mapping the page. DONE.**
2. What happens if there is a page fault on a mispredicted branch (i.e. the branch is taken) caused by the "not taken" prediction, and the address happens to be the first page past the end of the code space? **Can't happen (now). Needed to modify the implementation document. DONE.**
3. Clarify the definition of saved interrupt state (IRR, ICR, maybe others). Is it per context? Even though these are kernel only registers? How does that work exactly? **Not per context. Kernel special registers are unique, not per context.**
4. Similarly, have to decide about the processor status word (CPU flags) and interrupts. Are there really two of them? Or do they **always** have to be saved by interrupt handlers and restored? **Tentatively: there is one processor status word for usermode containing only the four flag bits, and a separate one for kernel mode containing four flag bits plus the T and IE bits. This means the kernel doesn't have to instantly save the usermode flags register, but it does have to save it when it's going to switch processes, unlike almost all the other state. It would be better to have flags per user context, but it may be very difficult, requiring a whole separate 3.3v RAM, etc.**
5. To put it more simply, there isn't a way to set the condition codes. Writing to the special register is possible but it takes two instructions and requires a register. **Added SCF and CCF. There's no easy way force set or reset on the others. DONE.**
6. Add the fault address to the saved state at least in the cases where it's relevant. **Added IDR, Interrupt Data Register. DONE.**
7. Possibly change the vectors (again!) so that "sys 0" through "sys 7" are vectors 8..15. This doesn't need to be documented for "user" documentation but it would need to be documented in a distinct "system architecture" document, if there was one. The issue with having user vector 0..7 be system vectors 0..7 is that vector 0 is hardwired to be the reset vector. **DONE**.
8. Need a way to read and to set the interrupt enable bit. Also, the Trap bit was probably a good idea. **DONE**.
9. Consider having the LVC and SVC instructions just set a bit in the ICR to indicate a fault occurred.