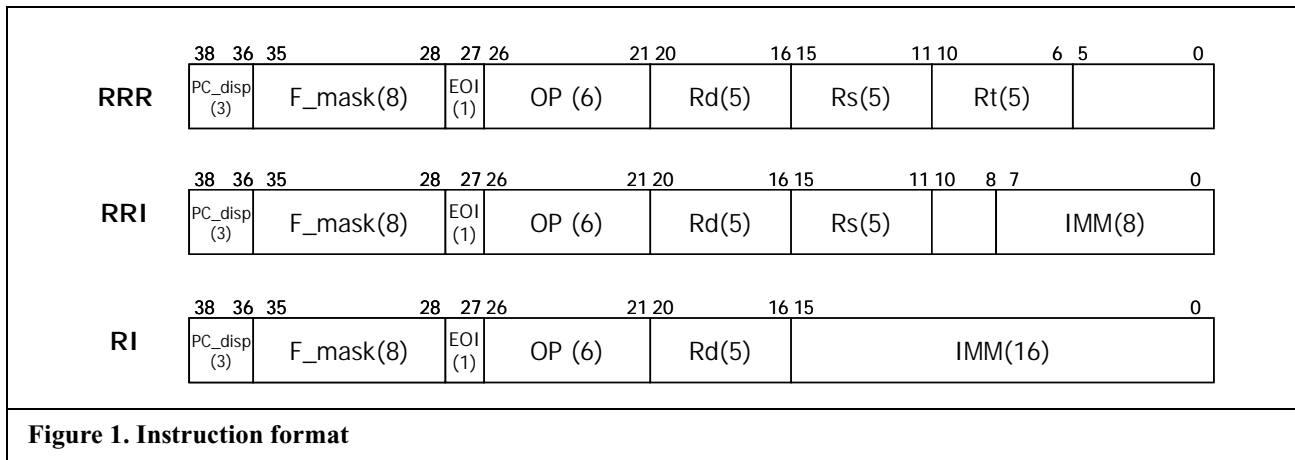


1.0 Micro-ISA

1.1 Instruction word format



1.2 Instruction word fields descriptions

Table 1: Field descriptions

Field	Description																
PC_disp (3)	PC update information. When an instruction w/ non-zero PC_disp is decoded, the content of PC is updated to PC + PC_disp. Usually, PC_disp is the number of z80 instruction bytes, and only the first instruction if the translation group has non-zero values in this field; other instructions have 0 in this field.																
F_mask (8)	<div>Each bit in this field represents a corresponding condition bit as follows:</div> <table><tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td>S</td><td>Z</td><td>X</td><td>H</td><td>X</td><td>P/V</td><td>N</td><td>C</td></tr></table> <div>Each bit in this field determines if the corresponding condition bit will be updated by the new condition bit generated by the ALU operation of the current instruction. If the bit is 1, the corresponding field in F register is updated; otherwise, the corresponding bit is not affected.</div> <div>As defined in F register, bit 5 and 3 are unused.</div>	7	6	5	4	3	2	1	0	S	Z	X	H	X	P/V	N	C
7	6	5	4	3	2	1	0										
S	Z	X	H	X	P/V	N	C										
EOI (1)	End of translation group, wich represents the z80 instruction boundary. This bit controls control transfer. Interrupts can occur only after the instruction with this field, or after a taken jump / branch.																
OP (6)	Instruction opcode																
Rd (5)	Destination register field. In RI format, this field can represent a source register																
Rs (5)	Source register 1 field (left-hand side operand in instruction descriptions)																
Rt (5)	Source register 2 field (right-hand side operand in instruction descriptions)																
IMM (8)	8-bit immediate (signed) field. If the instruction requires a 16-bit value from this field, it is sign-extended.																
IMM (16)	16-bit immediate (signed) field.																

1.3 Special-purpose registers

Table 2: Special purpose registers

Register	Description
archPC (16)	Architected 16-bit program counter. Incremented by PC_disp every clock cycle, or updated with the target PC address if a control transfer occurs. Note on CTIs (control transfer instructions): When the first instruction in the translation group, archPC is updated by PC_disp, pointing to the next sequential PC. When archPC should be updated by a taken relative branch instruction, you may assume that the offset field of the branch has been pre-calculated so that you don't have to worry about whether archPC is currently pointing to the current PC or the next PC
IFF1 (1)	determines if an incoming maskable interrupt is blocked. The content of this field is updated by EI and DI instructions. See z80 user's manual
IFF2 (1)	contains the previous state of IFF1. The content of this field is updated by EI and DI instructions. See z80 user's manual

1.4 General purpose registers

General purpose registers are accessed through the register file using 5-bit register specifiers. If the register identifier starts with 0, it is in 8/16-bit group, in which all 8-bit values are sign-extended before fed into the ALU. 16-bit group register identifiers always start with 1. In addition to z80 architected registers, this group has 4 temporary registers, which always end with 10. The order of register specifiers were determined by the register indexing scheme used in z80 architecture, in favor of the ease of translation. So don't complain that the order is so messy.

Table 3: General purpose registers

8-bit register group			16-bit register group		
register	specifier	description	register	specifier	description
R0	00000	zero register that always contains 0. Can be used as a 16-bit register	SP	11001	z80 stack pointer
A	01111	z80 accumulator register	AF	11100	z80 {A,F}
F	01101	z80 condition code vector			
B	00001	z80	BC	10000	z80 {B,C}
C	00011	z80			
D	00101	z80	DE	10100	z80 {D,E}
E	00111	z80			
H	01001	z80	HL	11000	z80 {H,L}
L	01011	z80			
I	00100	z80	IX	10001	z80
R	01000	z80	IY	10101	z80
T0	00010	8bit temp	T10	10010	{T1,T0}
T1	00110	8bit temp			
T2	01010	8bit temp	T32	11010	{T3,T2}
T3	01110	8bit temp			
			AF'	11111	z80 shadow AF
			BC'	10011	z80 shadow BC
			DE'	10111	z80 shadow DE

Table 3: General purpose registers

8-bit register group			16-bit register group		
register	specifier	description	register	specifier	description
			HL'	11011	z80 shadow HL

1.5 Instructions

IMPORTANT: Make sure that you are realizing flag bits correctly. In the table below, each instruction has an column “flags generated”, in which you can find a note such as “see z80 DAA”. Then, look up the table “flag notations” and find the detailed descriptions that you need to implement.

Table 4: Operand notation

notation	description
Rx	8bit register
RRx	16bit register
RRRx	8bit or 16bit register
IMM8	8-bit immediate (in RRI format). You may want to do sign-extension for this field if the actual computation is performed in 16-bit
IMM16	16-bit immediate (in RI format)

Table 5: Flag notations

	Description	S (sign)	Z (zero)	H (half carry / borrow)	P/V (parity / overflow)	N (negate)	C (carry / borrow)
standard (8bit)	standard 8bit add/sub	result (7)	result == 0 ? 1 : 0	ADD: carry at bit 3 to 4 SUB: borrow from bit 4	overflow at 8bit computation	ADD: 0 SUB: 1	ADD: carry at bit 7 to 8 SUB: borrow from bit 8
standard(16bit)	standard 16bit add/sub	result (15)	result == 0 ? 1 : 0	ADD: carry at bit 11 to 12 SUB: borrow from bit 12	overflow at 16bit computation	ADD: 0 SUB: 1	ADD: carry at bit 15 to 16 SUB: borrow from bit 8
z80 AND		result (7)*	result(7:0) == 0 ? 1 : 0*	1	overflow at 8bit computation**	0	0
z80 OR		result (7)*	result(7:0) == 0 ? 1 : 0*	0	overflow at 8bit computation**	0	0
z80 XOR		result (7)*	result(7:0) == 0 ? 1 : 0*	0	1 when even parity	0	0
z80 shift/rotate for XXX	for z80 RLC, RRC, RL, RR, SLA, SRA, SRL	result (7)*	result(7:0) == 0 ? 1 : 0*	0	1 when even parity	0	consult z80 user's manual for carry flags
z80 shift/rotate for XXXA	for z80 RLCA, RRCA, RLA, RRA	not affected***	not affected***	0	not affected***	0	consult z80 user's manual for carry flags
z80 RRD/RLD		result (7)	result == 0 ? 1 : 0	0	1 when even parity	0	not affected***
z80 bit b		don't care	result == 0 ? 1 : 0	1	don't care	0	not affected***

Table 5: Flag notations

	Description	S (sign)	Z (zero)	H (half carry / borrow)	P/V (parity / overflow)	N (negate)	C (carry / borrow)
z80 in		bit 7 of the data from I/O	the data from I/O == 0 ? 1:0	0	1 when the data from I/O has even parity	0	not affected***
z80 DAA***		A (7)	A == 0 ? 1:0	see z80 DAA instruction	1 when A has even parity	not affected***	see z80 DAA instruction
z80 CPL		not affected***	not affected***	1	not affected***	1	not affected***

* Instructions may execute with 16-bit registers. However, sign and zero flags are generated by examining only result bits (7:0).

**In z80 manual, it is unclear that overflow bit for AND and OR operation. Just follow the standard boolean equation for overflow checking.

*** it will be taken care of by annotation bits. Therefore, treat it as “don’t care”

**** follow flag bits in z80 user’s manual

Table 6: Instructions

type	Opcod e	Synopsis	for- mat	Semantic	description	Flags gener- ated	note
ALU arith- metic	00	add Ra, Rb, Rc	RRR	$Ra = Rb + Rc$		standard (8bit)	
	01	addi Ra, Rb, IMM8	RRI	$Ra = Rb + IMM(\text{signed})$		standard (8bit)	
	02	sub Ra, Rb, Rc	RRR	$Ra = Rb - Rc$		standard (8bit)	
	03	add16 RRa, RRb, RRc	RRR	$RRa = RRb + RRc$		standard (16bit)	
	04	addi16 RRa, RRb, IMM8	RRI	$RRa = RRb + IMM(\text{signed})$		standard (16bit)	
	05	sub16 RRa, RRb, RRc	RRR	$RRa = RRb - RRc$		standard (16bit)	
	1A	subi Ra, Rb, IMM8	RRI	$Ra = Rb - IMM(\text{signed})$		standard (8bit)	newly added
	1B	subi16 RRa, RRb, IMM8	RRI	$RRa = RRb - IMM(\text{signed})$		standard (16bit)	newly added

Table 6: Instructions

type	Opcod e	Synopsis	for- mat	Semantic	description	Flags gener- ated	note
ALU logi- cal	06	and RRRa, RRRb, RRRc	RRR	RRRa = bitwise RRRb & RRRc		follow z80 AND	
	07	andi RRRa, RRRb, IMM8	RRI	RRRa = bitwise RRRb & IMM8		follow z80 AND	If RRR is a 16bit register, IMM8 is sign extended
	08	or RRRa, RRRb, RRRc	RRR	RRRa = bitwise RRRb RRRc		follow z80 OR	
	09	ori RRRa, RRRb, IMM8	RRI	RRRa = bitwise RRRb IMM8		follow z80 OR	if RRR is a 16 bit register, IMM8 is sign extended
	0A	xor RRRa, RRRb, RRRc	RRR	RRRa = bitwise RRRb xor RRRc		follow z80 xor	
	0B	xori RRRa, RRRb, IMM8	RRI	RRRa = bitwise RRRb xor IMM8		follow z80 xor	if RRR is a 16bit register IMM8 is sign extended
	0C	not RRRa, RRRb	RRI	RRRa = bitwise ~RRRb		follow z80 CPL	
	0D	shiftrotate Ra, Rb, OP	RRI	OP field is defined as fol- lows: RLC: 00 RLCA: 01 RRC: 02 RRCA: 03 RL: 04 RLA: 05 RR: 06 RRA: 07 SLA: 08 SRA: 09 SRL: 0A SLL: 0B	direct imple- mentation of rotate and shift z80 instruc- tions	as defined in z80 shift and rotate instruc- tions	Even in ???A instructions, Ra and Rb will be explicitly set to A register by the xla- tor. Implicitly, this instruc- tion assumes to have F register as an input note: SLL (shift left logical) operation is undocu- mented and will be added. please find some informa- tion on undocumented z80 instruction
	0E	get4 Ra, Rb, IMM8	RRI	Ra(3:0) = Rb(7:4) if IMM8!=0, Rb(3:0) other- wise. Ra(7:4) = 0000	get H/L portion 4 bits	none	for RLD operation
	0F	merge44 Ra, Rb, Rc	RRR	Ra = {Rb(3:0), Rc(3:0)}	merge 2 L por- tion 4-bit chunks into one 8-bit value	follow RRD / RLD	for RLD operation
	10	mvPC RRa	RRI	RRa = PC	copy the cur- rent content of PC into RRa	none	This is required since PC is a special purpose regis- ter
	1C	mvIff RRRa, IMM8	RRI	RRRa(0) = iff1 if IMM8 = 1 RRRa(0) = iff2 if IMM8 = 2 other bits in RRRa become 0	Read the iff1 / iff2 into RRRa	none	undetermined if IMM8 is not 1 or 2 newly added
ALU spe- cial	11	DAA Ra, Rb	RRI	Read Rb, perform DAA operations and put the result into Ra same as z80 DAA	direct imple- mentation of z80 DAA instruction	follow z80 DAA	Ra and Rb will be explicitly set to A register by the xla- tor. Implicitly, this instruc- tion assumes to have F register as an input. special ALU support

Table 6: Instructions

type	Opcod e	Synopsis	for- mat	Semantic	description	Flags gener- ated	note
ALU bit operation	12	getbit Ra, Rb, IMM8	RRI	$Ra(0) = Rb(\text{bit IMM})$ $Ra(7:1) = 0000000$	get one bit from Rb, copy the bit into Ra, clears other bits in Ra	see bit b, (??)	8bit op, IMM is x00 ~ x07. the result undetermined if IMM is out of range
	13	ngetbit Ra, Rb, IMM8	RRI	$Ra(0) = \sim Rb(\text{bit IMM})$ $Ra(7:1) = 0000000$	get one bit from Rb, invert the bit, copy the bit into Ra, clears other bits in Ra	see bit b, (??)	8bit op, IMM is x00 ~ x07. the result undetermined if IMM is out of range
	14	setbit Ra, Rb, IMM8	RRI	$Ra(\text{bit IMM}) = (Rb \neq 0 ?$ $1 : 0)$ other bits in Ra not affected*	set one bit in Ra with the content of Rb	none	IMM is x00 ~ x07. the result undetermined if IMM is out of range
	15	nsetbit Ra, Rb, IMM8	RRI	$Ra(\text{bit IMM}) = (Rb == 0 ?$ $1 : 0)$ other bits in Ra not affected*	set one bit in Ra with the content of Rb	none	IMM is x00 ~ x07. the result undetermined if IMM is out of range
Control transfer	20	j IMM16	RI	$PC = IMM$	jump direct to IMM16	none	always taken
	21	jr RRa	RI	$PC = RRa$	jump direct to the content of RRa	none	always taken
	22	Jc RRRa, IMM16	RI	if RRRa is not 0, $PC =$ IMM	jump direct to IMM16 if RRRa is true (non zero)	none	conditional, RRRa is either 8 or 16 bits
	23	Jrc RRRa, RRb	RRI	if RRRa is not 0, $PC =$ RRb	jump direct to the content of RRb if RRRa is true	none	conditional, RRRa is either 8 or 16 bits
	24	bne RRRa, RRRb, IMM8	RRI	if $RRRa \neq RRRb$, $PC =$ $PC + IMM8$ (signed)	jump relative if ($RRRa ==$ $RRRb$)	none	relative branch, the offset is dependent on z80 ops, assuming PC has already been incremented at the beginning of the translated instruction group
	25	beq RRRa, RRRb, IMM8	RRI	if $RRRa == RRRb$, $PC =$ $PC + IMM8$ (signed)	jump relative if ($RRRa \neq$ $RRRb$)	none	relative branch, the offset is dependent on z80 ops, assuming PC has already been incremented at the beginning of the translated instruction group
load / store	30	ld Ra, (RRb + IMM8)	RRI	$Ra = [RRb +$ $IMM(\text{signed})]$		none	8bit load
	31	st Ra, (RRb + IMM8)	RRI	$[RRb + IMM(\text{signed})] =$ Ra		none	8bit store
	16	limm RRRa, IMM16	RI	$RRRa = IMM$	load 16-bit immediate into the register	none	higher 8bits of IMM are ignored if RRRa is Ra
Interrupt	17	EI	RRR	set iff to 1	enable inter- rupt	none	interrupt FF (iff1, iff2) con- trol. See z80 EI implemen- tation
	18	DI	RRR	set iff to 0	disable inter- rupt	none	interrupt FF (iff1, iff2) con- trol. See z80 DI implemen- tation
	19	IM IMM16	RI	set the interrupt mode to IMM16 (do nothing)		none	will not be implemented since SMS uses only inter- rupt mode 1.

Table 6: Instructions

type	Opcod e	Synopsis	for- mat	Semantic	description	Flags gener- ated	note
I/O	32	in Ra, (Rb + IMM8)	RRI	Ra = I/O port (Rb + IMM8)	I/O port read	follow z80 In	check z80 IN instruction for address bus controls
	33	out Ra, (Rb + IMM8)	RRI	I/O port (Rb + IMM8) = Ra	I/O port write	none	check z80 OUT instruction for address bus controls

* since unrelated bits in setbit / nsetbit are unmodified, the ALU should read the original content of the target register, change one bit, and overwrite all bits in the target register again.

2.0 Interface Definition

2.1 Signal handshaking

All signals defined in this section should transit synchronously with respect to the positive edge of the clock (CLK) signal. No input and output signal may depend on edge-triggered operations, which implies that all input signals are examined only at the positive edge of CLK, only by the input signal levels. Therefore, an outgoing signal generated by the source in the current clock cycle affects the output of the destination in the following clock cycle.

All input and output signals follow the rules:

- Detecting an incoming signal: level detection at positive edge of CLK
- generating an outgoing signal: start transiting at negative edge of CLK, sustaining until the next negative edge of CLK, if possible. If not possible, you should guarantee that the outgoing signal does not transit in sync with the positive edge of CLK.

Note that accessing memory interface (mem_arbit) unit follows different strategies to minimize the inter-communication delay, and the signal handshaking defined here does not apply.

2.2 Reset handling

All modules have a reset port to initialize their states and values. The reset is synchronously detected by the level of the signal at positive edges of CLK. Edge detections of the reset is depreciated. The recommended reset implementation is shown as follows:

```
always@(posedge CLK) // do not put RST here
begin
    if (RST == 1) begin // detect the level synchronously
        // put reset routine here
    end
    else // other routines
end // end of always routine
```

Figure 2. Reset implementation

2.3 Translator -- RISC core

The following table defines signals between the translator and the RISC core.

Table 7: Interface between translator and RISC core

Signal name	bits	direction (xlator -- core)	Description
Iword	39	—>	Instruction word (instruction + annotation), as defined in Figure 1. Note: there is no such signal as “no instruction” or “instruction queue empty”. Instead, the instruction queue always generate nop w/ EOI if no instruction is available from the queue. Therefore, the RISC core can assume that fetch stall does not occur.
pipe_stall	1	<—	Assert when the RISC core should not dequeue an instruction from the instruction queue (xlator) due to pipeline stall. The pipeline stall condition occurs because of structural hazard and memory wait.
flush	1	<—	Assert when a control tranfer (taken branch / jump) or a memory paging should occur. Note that interrupts use <i>interrupt</i> signal to notify xlator. When xlator receives this signal, xlator sets <i>fetchPC</i> to the content of <i>targetPC</i> , and invalidates all instructions in instruction queue, xlator module and instruction bytes in z80 decoder. Typically, <i>fetchPC</i> in xlator is synchronized with <i>archPC</i> by this signal, and the xlator starts fetching again from the new <i>fetchPC</i> (updated by <i>targetPC</i>) after flush. RISC core: when this signal is asserted, targetPC should contain the correct target PC address that updates archPC and fetchPC.
INT	1	<—	Assert by the interrupt handler when an maskable interrupt should be serviced. The xlator and the RISC pipeline are also flushed as if <i>flush</i> is asserted, but xlator starts generating a serie of u-ops that save <i>archPC</i> onto the stack in memory, and that jump to the trap location.
NMI	1	<—	Assert by the interrupt handler when a non-maskable interrupt should be serviced. The xlator and the RISC pipeline are also flushed as if <i>flush</i> is asserted, but xlator starts generating a serie of u-ops that save <i>archPC</i> onto the stack in memory, and that jump to the trap location.
targetPC	16	<—	Valid only when <i>flush</i> is 1; ignored otherwise. It is the RISC core's responsibility to put an appropriate target PC for taken branch / jump (but trap locations are set up by xlator)

3.0 Handling control transfers -- not done yet

3.1 Interrupt handling

When an interrupt (maskable / non-maskable) is detected by the RISC core,

1) the RISC core determines whether the processor should handle the interrupt. If it is maskable and *iff1* is 0, the interrupt request is ignored; else the request will be processed. Reset *iff1* after the reception to prevent other incoming interrupts.

2) The interrupt handler in RISC core holds the interrupt signal until an instruction with EOI is encountered in decode/RF stage, or a control transfer occurs. The *archPC* has already been updated with the next PC, which may be sequential or a taken target.

3) The interrupt handler asserts *interrupt* signal as well as *targetPC* (a trap location, depending on whether maskable / non-maskable) to xlator. Then, f