1 .0 Translator Part

1.1 Overview

1.2 Module definitions

1.2.1 uop_queue

uop_queue is a FIFO queue that receives up to 4 micro instructions from the translator (xlator), and sends 1 micro instruction to the RISC core every clock cycle. If the queue is empty but there is a dequeue request from the RISC core, uop_queue generates nops with EOI 1.

Table 1: uop_queue signals

Signal	Bits	In/out	Description	Connections
CLK	1	IN	clock	
RST	1	IN	Active high, synchronous	
flush	1	IN	Flush all instructions in the queue and make it empty	from RISC (*flush*, *interrupt*)
IN_uop0~3	39	IN	uops (0 ~ 3) generated from xlator	from xlator
WR_EN0~4	1	IN	write enable (0 ~ 3) for each instruction, indicating that instructions to be enqueued are ready..	from xlator
pipe_stall	1	IN	when asserted, no instrcution is dequeued, and **OUT_uop does not change* (same as the previous clock cycle)** If not asserted, uop_queue dequeues an instruction.	from RISC (*pipe_stall)*
OUT_uop	39	OUT	uop dequeued. If no instruction is available, a nop w/ EOI is generated	to RISC
Q_full	1	OUT	indicates that the queue is full. The current write requests are ignored.	to xlator (Q_full)

* uop_queue works as a pipeline latch between xlator and the decode/RF stage. OUT_uop should not change and be same as the previous clock cycle in order not to lose any instruction when a pipeline stall occurs.

1.2.2 z80_fetcher

z80_fetcher sends an read request to memory so that the pre-decoder reads an instruction byte. If the operation was successful, z80_fetcher increments *fetchPC* by 1 and repeats the operation. If *flush* is asserted, *fetchPC* is updated with *targetPC*.

Table 2: z80_fetcher signals

Signal	Bits	In/out	Description	Connections
CLK	1	IN	clock	
RST	1	IN	Active high, synchronous. *fetchPC* is cleared to 0.	
flush	1	IN	when asserted, fetchPC is updated with the content of *targetPC*	from RISC (*flush*)
MREQ	1	OUT	read request to the memory interface	to memory interface (I_MREQ)
I_Addr	16	OUT	instruction address generated from *fetchPC*	to memory interface (I_addr)
I_wait	1	IN	if asserted, *fetchPC* is not incremented	from memory inter-face (I_wait)

Table 2: z80_fetcher signals

Signal	Bits	In/out	Description	Connections
z80dec_stall	1	IN	if asserted, *fetchPC* is not incremented	from z80_decoder (z80dec_stall)
targetPC	16	IN	*fetchPC* is updated with the content of this signal when *flush* is asserted.	from RISC (targetPC)

1.2.3 z80_decoder.

Table 3: z80_decoder signals

Signal	Bits	In/out	Description	Connections
CLK	1	IN	clock	
RST	1	IN	Active high, synchronous.	
flush	1	IN	invalidate instructions in the decoder and ignore I_byte	
I_wait	1	IN	ignore I_byte if this is asserted	from memory
z80dec_stall	1	OUT	assert this if (I_word_rdy && xlator_stall)	to z80_fetcher
I_byte	8	IN	read a byte from this port if *I_wait* is false	from memory
I_word	32	OUT	z80 instruction word	to xlator
I_word_rdy	1	OUT	z80 instruction word / decode info ready	to xlator
z80_I_ID	8	OUT	z80 instruction identifier, used in translation	to xlator
xlator_stall	1	IN	xlator stall (conversely RD_EN_ from xlator when I_word_rdy is asserted)	from xlator

1.2.4 xlator

Table 4: xlator signals

Signal	Bits	In/out	Description	Connections
CLK	1	IN	clock	
RST	1	IN	Active high, synchronous.	
flush	1	IN	invalidate instructions in xlator, and reset the state	
uop0~3	39	OUT	uops (0 ~ 3)	to uop_queue
WR_EN0~3	1	OUT	indicating translated instructions are ready at each uop (0 ~ 3) ports	to uop_queue
NMI	1	IN	if *NMI* or *INT* is asserted, the current translation being processed in xlator is canceled, and xlator generates a sequence of instructions to save *archPC* onto the stack, jump to a trap located (determined by the type of interrupt). Meanwhile, xlator_stall is asserted so that no z80 instructions are read from the memory before the control is transfered to the trap location.	from interrupt handler in RISC core
INT	1	IN		from interrupt handler in RISC core
xlator_stall	1	OUT	xlator stall, indicating that there is a pending translation	to z80_decoder
I_word	32	IN	z80 instruction word	from z80_decoder
I_word_rdy	1	IN	z80 instruction word / decode info is ready	from z80_decoder
z80_I_ID	8	IN	z80 instruction identifier, used in translation	from z80_decoder
Q_full	1	IN	indicating that uop_queue is full. Since the current write requests are ignored, uop0~3 and WR_EN0~3 should be sustained.	from uop_queue

2 .0 RISC Core Part

2.1 Module design strategy

The time is always against us, and it is likely that we are not going to achieve good designs in time.

Therefore, all modules will have many redundant inputs/outputs, and make each module decide what operation it will perform, as opposed to feeding necessary control signals into it.

2.2 Design overview

2.2.1 RF stage

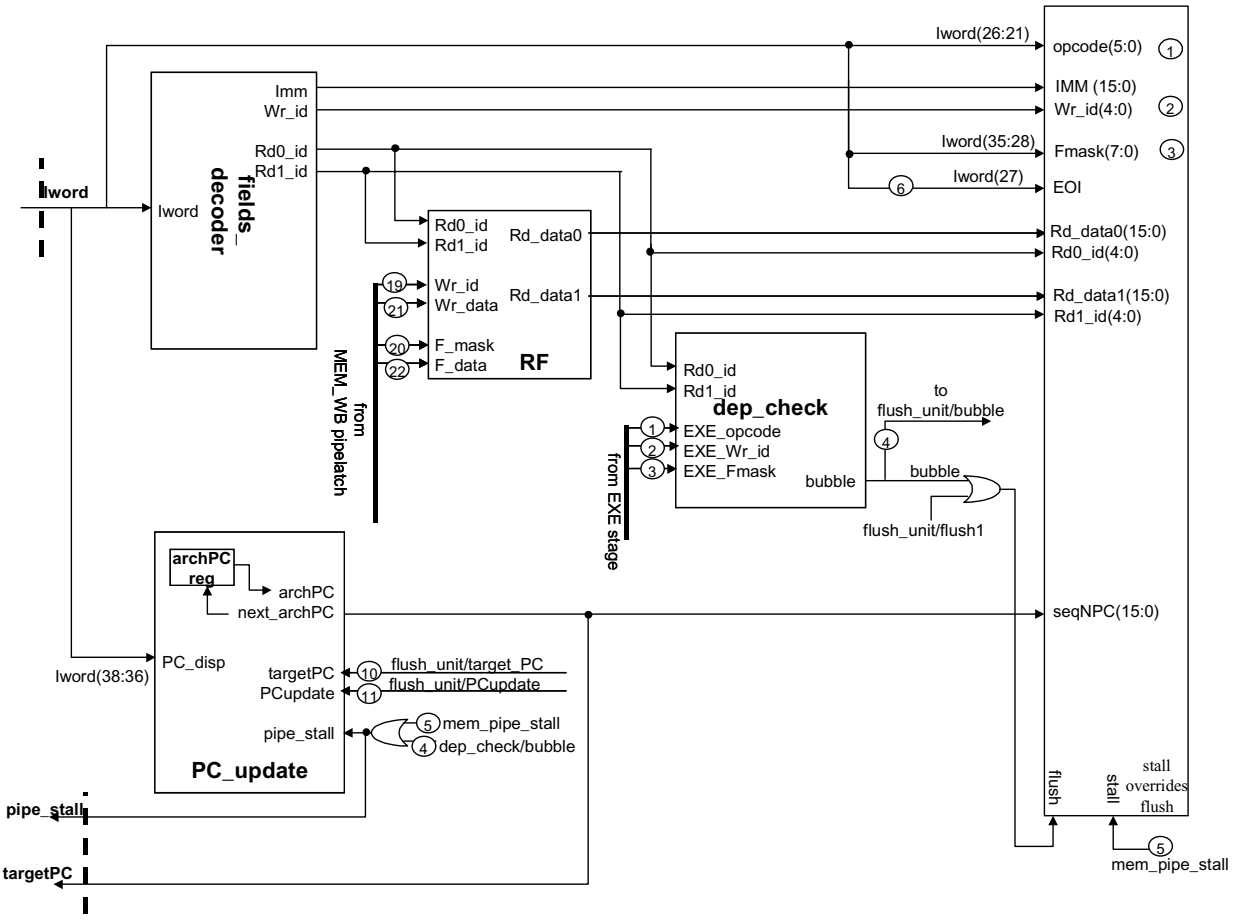

Figure 1. RF stage

2.2.2 EXE stage

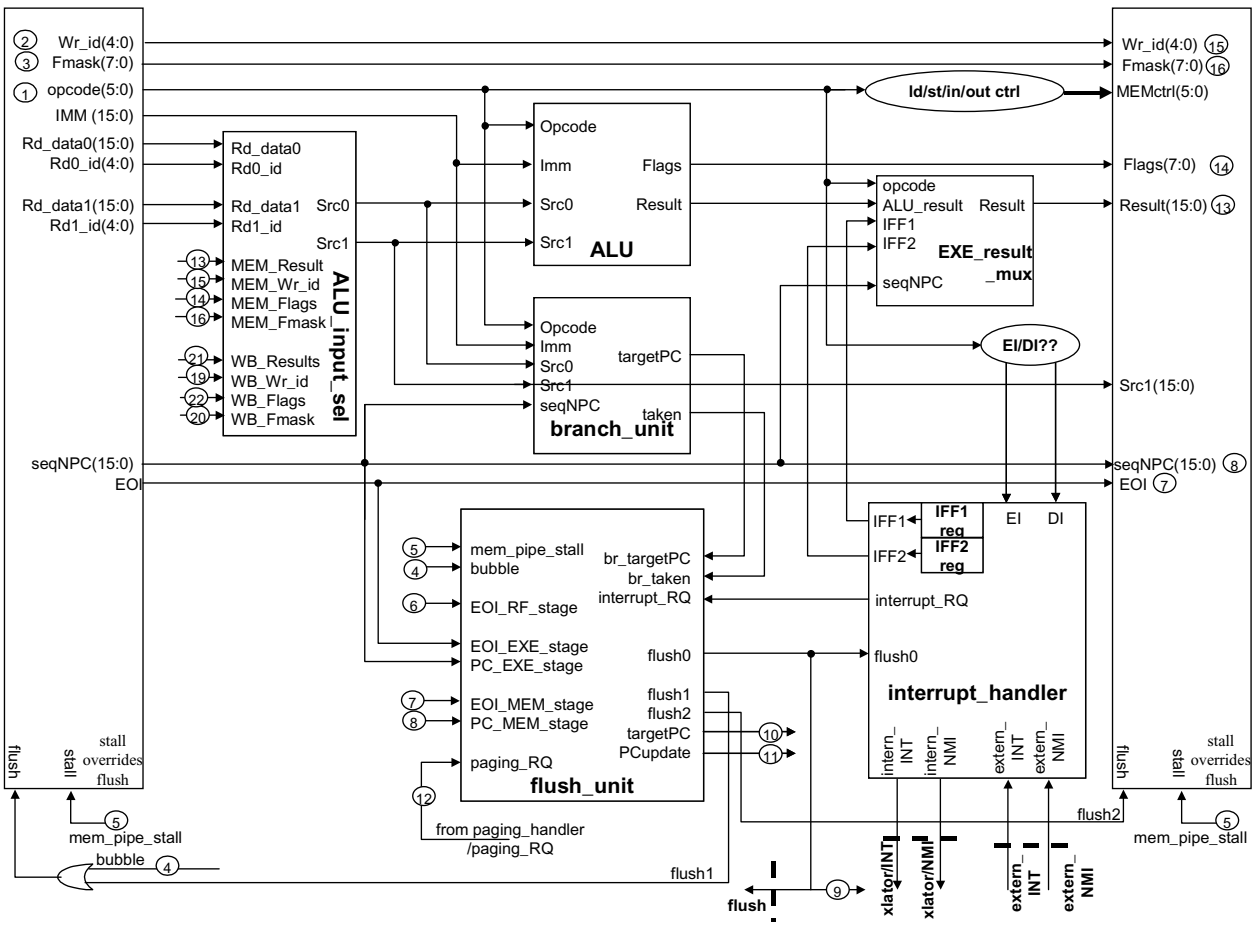

Figure 2. EXE stage

2.2.3 MEM stage

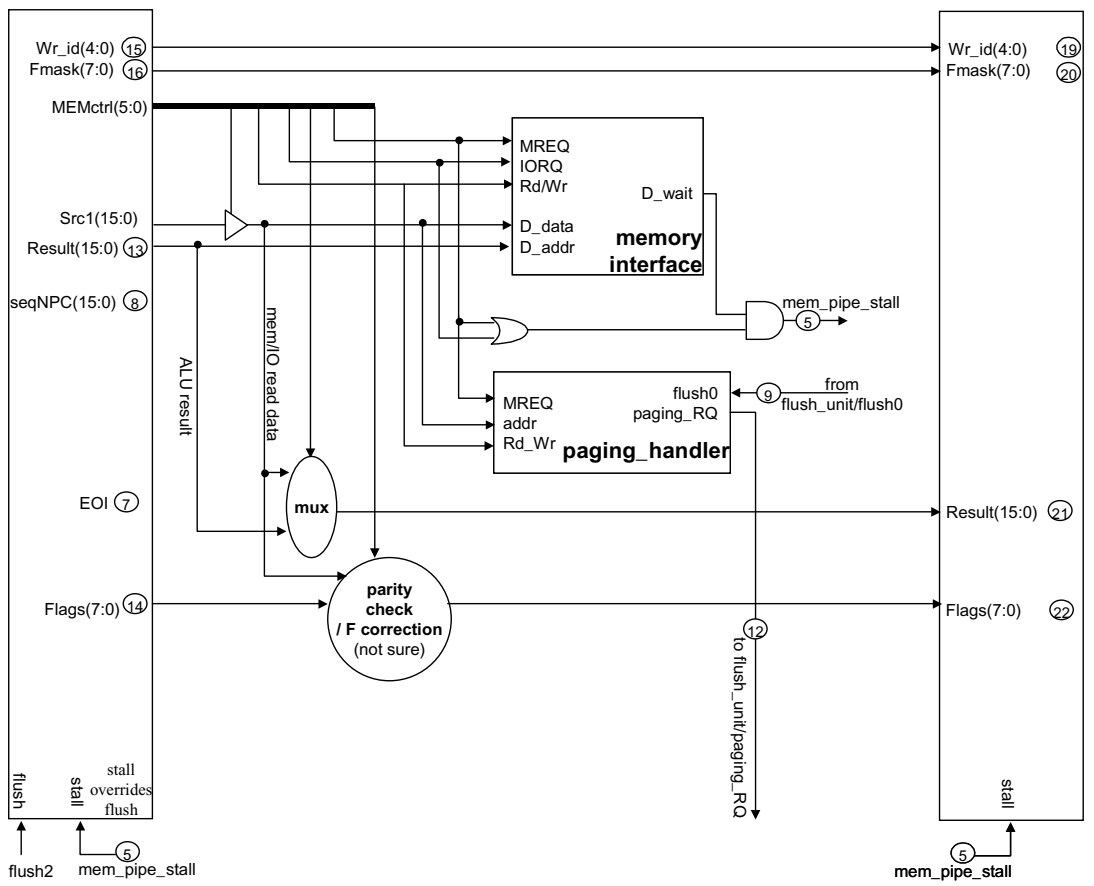

Figure 3. MEM stage

2.3 Module definitions

2.3.1 ALU

ALU handles arithmetic, logical, special, bit, load/store and I/O operations. ALU should be purely combinational.

All input and output values to ALU are always 16 bits even for 8-bit operations. All input values should be sign-extended to 16 bits if they comes from 8-bit values. When the 16-bit result from a ALU operation is written back to RF, the higher 8 bits will be cut off if the target register is 8-bit. Therefore, implementations for 8-bit ALU operations guarantees to be equivalent to reading 8-bit inputs and generating 8-bit outputs.

For example, the result value from *add* and *add16* should be the same given same inputs (but flag bits should differ). However, the result value from *add* will be tailored to 8 bits when written back to RF. Since the inputs (8 bits) have been sign-extended to 16 bits, the final result is equivalent to add 8-bit values on a 8-bit ALU. It should be guaranteed that the correct flags are generated with regard to 8/16-bit operations. E.g. In 8-bit operations with 8-bit standard flag definitions, Z flag is generated by examining only result(7:0), ignoring result (15:8)

Which sources among Src0, Src1 and Imm are used for the computation is determined by Opcode.

Table 5: ALU

Signal	Bits	In/out	Description	Connections
Src0	16	IN	16-bit source value0	
Src1	16	IN	16-bit source value1	
Imm	16	IN	16-bit source value from IMM field. IMM8 should be sign-extended before connected. This field may work as either a source value, or as a part of opcode i.e. as in *shiftrotate* (ALU determins this 16-bit field will be used as 16 bits or 8 bits by looking at Opcode).	
Opcode	6	IN	instruction opcode. This field (sometimes combined with Imm) determines which type of operation should be performed. For load/store or I/O operations, ALU performs AGENs	
Flags	8	OUT	Condition code outputs	
Result	16	OUT	16-bit result value	

Note that compare operations for branches are performed in a separate unit.

Outputs for instructions that the ALU does not support are unpredictable.

The following table defines instructions supported by the ALU and the port mapping. In this table, port *Src0*, *Src1* and *Imm* implies that the ALU can read the 16-bit register / immediate values (defined in synopsis) from the ports specified.

Table 6: ALU instructions & port mapping

type	Opcode	Synopsis	Module	port Src0	port Src1	port Imm	port Result	note
ALU arithmetic	00	add Ra, Rb, Rc		Rb	Rc		Ra	
	01	addi Ra, Rb, IMM8		Rb		IMM	Ra	
	02	sub Ra, Rb, Rc		RRb	RRc		Ra	
	03	add16 RRa, RRb, RRc		RRb	RRc		RRa	
	04	addi16 RRa, RRb, IMM8		RRb		IMM8	RRa	
	05	sub16 RRa, RRb, RRc		RRb	RRc		RRa	
	1A	subi Ra, Rb, IMM8		Rb		IMM8	Ra	
	1B	subi16 RRa, RRb, IMM8		RRb		IMM8	RRa	
ALU logical	06	and RRRa, RRRb, RRRc		RRRb	RRRc		RRRa	
	07	andi RRRa, RRRb, IMM8		RRRb		IMM8	RRRa	
	08	or RRRa, RRRb, RRRc		RRRb	RRRc		RRRa	
	09	ori RRRa, RRRb, IMM8		RRRb		IMM8	RRRa	
	0A	xor RRRa, RRRb, RRRc		RRRb	RRRc		RRRa	
	0B	xori RRRa, RRRb, IMM8		RRRb		IMM8	RRRa	
	0C	not RRRa, RRRb		RRRb			RRRa	
	0D	shiftrotate Ra, Rb, OP		Rb	F	OP	Ra	ALU reads carry bits from port Src1 (F register). For z80 instructions w/ a target reg of A by definitions (RLCA....), xlator will explicitly setup A as a target.
	0E	get4 Ra, Rb, IMM8		Rb		IMM8	Ra	
	0F	merge44 Ra, Rb, Rc		Rb	Rc		Ra	

Table 6: ALU instructions & port mapping

type	Opcode	Synopsis	Module	port Src0	port Src1	port Imm	port Result	note
ALU special	11	DAA Ra, Rb		Rb	F		Ra	ALU reads carry bits from port Src1 (F register). Both source / target A will be explicitly specified by xlator.
ALU bit operation	12	getbit Ra, Rb, IMM8		Rb		IMM8	Ra	
	13	ngetbit Ra, Rb, IMM8		Rb		IMM8	Ra	
	14	setbit Ra, Rb, IMM8		Rb	Ra	IMM8	Ra	Ra on Src1 for read-modified-write operations
	15	nsetbit Ra, Rb, IMM8		Rb	Ra	IMM8	Ra	Ra on Src1 for read-modified-write operations
Address calculation	30	ld Ra, (RRb + IMM8)		RRb		IMM8	Address	perform add operation the result from this operation goes to MEM/Addr
	31	st Ra, (RRb + IMM8)		RRb	(Ra)	IMM8	Address	perform add operation the result from this operation goes to MEM/Addr. Ra is not used for ALU but the value is accessed from RF
	16	limm RRRa, IMM16				IMM16	RRRa	
I/O	32	in Ra, (Rb + IMM8)		Rb		IMM8	Address	perform add operation the result from this operation goes to MEM/Addr
	33	out Ra, (Rb + IMM8)		Rb	(Ra)	IMM8	Address	perform add operation the result from this operation goes to MEM/Addr. Ra is not used for ALU but the value is accessed from RF

2.3.2 Register File

General purpose registers are accessed through the register file. Any updates to an entry are automatically reflected to corresponding entries. Read operations are not synchronous: changes in read register specifiers are directly reflected to outputs. However, write operations are synchronous wrt posedge of CLK. The register file supports bypass paths that bridge between write and read operations in a clock cycle. Logically, this is equivalent to writing a value in the first half of the clock period, and read a value for the other half of the clock period. Register specifier, structure definitions and port specifications are as follows:

Table 7: 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			

Table 7: General purpose registers

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

Table 8: RF

Signal	Bits	In/out	Description	Connections
CLK	1	IN	posedge CLK	
RST	1	IN	synchronous reset	
Rd0_id	5	IN	Read register specifier 0	
Rd1_id	5	IN	Read register specifier 1	
Rd_data0	16	OUT	Read data out 0. When a 8-bit entry is accessed, the value is sign-extended	
Rd_data1	16	OUT	Read data out 1. same as above	
Wr_id	5	IN	Write register specifier Note: since RF does not have write enable signals, a write operation is disabled if this field is 00000.	
Wr_data	16	IN	Write data in. When the target is 8-bit entry, Wr_data(15:8) is cut off internally before being stored	
Fmask	8	IN	Flag update mask. see microISA definition	
F_data	8	IN	Flag data to be written	

2.3.3 Fields decoder

This module is pure combinational logic. It reads the whole instruction word, looks at the opcode, and generate 2 source identifier for RF read ports, 1 destination identifier for the RF write port, and a 16-bit immediate value from instruction word bit fields. 00000 is generated for idle ports. 16’b0 is generated for unused immediate value. The destination identifier (target register) for a RF write port will make a round trip back to the RF in writeback stage for a store oper-

ation. Also, source identifiers are passed to next pipeline stages for bypass modules

Table 9: fields_decoder

Signal	Bits	In/out	Description	Connections
Iword	39	IN	full instruction word	
Rd0_id	5	OUT	register identifier for Rd0 in RF. 00000 is generated if the port is unused.	to RF/Rd0_id
Rd1_id	5	OUT	register identifier for Rd1 in RF. 00000 is generated if the port is unused.	to RF/Rd1_id
Imm	16	OUT	If the instruction is RI type, this field is Iword[15:0]. If it is RRI type, this field is the sign-extended Iword[7:0] If the instruction does not require an immediate value, this field is 0.	to ALU/Imm & branch_unit/Imm through the pipe latch
Wr_id	5	OUT	register identifier for Wr in RF. 00000 is generated if the write port is unused.	

*Shiftrotate*and *DAA* instructions may require the condition code for their computations. Although these instructions are RRI type and has only 1 source operand (Rd0_id), the instructions in fact assume F register as a source operand implicitly. Therefore, this module generates F register ID (01101) in Rd1_id to read F register needed for their computations.

2.3.4 ALU input selector (bypass logic)

This module is located right before the decode-execution pipe latch. It determines which values (from RF or ALU/Result or ALU/Flags or Mem/Rd_data or Mem/Flags) will be fed into ALU inputs (and branch_unit also). Pure combinational logic.

Due to RF configurations, this module should implement partial bypass from 8/16bit results to 16/8bit source, such as L <- HL or HL <- H. In case of ??bit to 8bit register forwardinig, the forwarded value should be sign-extended before put on ouput ports, In addition, this module also allows bit-wise bypass from previous Flag values, considering Fmask.

Table 10: ALU_input_sel

Signal	Bits	In/out	Description	Connections
Src0	16	OUT	selected source value0	to ALU/Src0 through the pipe latch
Src1	16	OUT	selected source value1	to ALU/Src1through the pipe latch
Rd_data0	16	IN	source value candidate from RF	from RF/Rd_data0
Rd0_id	16	IN	source0 ID from RF	from RF/Rd0_id
Rd_data1	16	IN	source value candidate from RF	from RF/Rd_data0
Rd1_id	16	IN	source1 ID from RF	from RF/Rd1_id
MEM_Result	16	IN	source value candidate from MEM	from ALU's result in MEM
MEM_Wr_id	5	IN	target register ID of the MEM_Result	from Wr_id in MEM-stage
MEM_Flags	8	IN	source value candidate (flag data) from MEM	from MEM/Flags

Table 10: ALU_input_sel

Signal	Bits	In/out	Description	Connections
MEM_Fmask	8	IN	flag mask of MEM_Flags	from Fmask in MEM stage.
WB_Result	8	IN	source value candidate from WB	from the result value in WB stage
WB_Wr_id	5	IN	target register ID of the WB_Result	from Wr_id in WB stage
WB_Flags	8	IN	source value candidate (flag data) from WB	from Flags in WB stage
WB_Fmask	8	IN	flag mask of WB_Flags	from Fmask in WB stage

Implementation note on bypass / forwarding:

- Two bypass paths are installed between:

  From the beginning of MEM stage (ALU output) to the beginning of EXE stage.

  From the beginning of WB stage (memory read data / ALU output) to the beginning of EXE stage.
- If dependences from both mem and execution stages are detected, values in execution stage has higher priority than those in memory stage in forwarding.
- Parity setting in *IN* instruction should be handled. Let's forget about this problem for now.
- We don't implement memory to memory bypass paths (memory move operations implemented by a series of load and stores). Instead, the control logic artificially injects pipe stalls to prevent these conditions.
- MEM_Result is a out from the mux between the memory read data and ALU output.
- *IN* and *ld* instructions do not generate results in the EXE stage, and this unit may incorrectly decide to forward the ALU result (agen result). However, this is also the bubble condition detected by dep_check module, which makes the current incorrect forwarding invalid. Therefore, the case does not happen.
- Forwarding data merging may occur. i.e. an instruction in WB changes H, another instructiono in MEM changes L, and the instruction in EXE reads HL. This may happen as a bit-wide merging operation in F register.

2.3.5 Branch unit

Pure combinational logic. This module generates target PC values for branches / jumps, computes conditions and determines if the control flow changes. Although the condition checking can be done in ALU rather than in this unit, I decided that this unit takes care of everything since all instructions do just simple equality checks, and do not require arithmetic operations (like less than or greater...)

Table 11: branch_unit

Signal	Bits	In/out	Description	Connections
Src0	16	IN	16-bit source value0	same as ALU/Src0
Src1	16	IN	16-bit source value1	same as ALU/Src1
Imm	16	IN	16-bit immediate value. Used to generate targetPC	same as ALU/Imm
Opcode	6	IN	opcode, determines how to generate PC and conditions	same as ALU/Opcode
seqNPC	16	IN	next sequential PC, which has been updated by PC_disp	*from*

Table 11: branch_unit

Signal	Bits	In/out	Description	Connections
targetPC	16	OUT	seqNPC + branch offsets from Iword for branches, or targetPC for jumps, regardless of taken/untaken	
taken	1	OUT	true if the branch / jump is taken because the condition is true or it is always taken	

2.3.6 Flush unit

pure combinational logic. The flushing unit determines the followings:

- from which stage the instruction should be flushed: 3 flushing locations. *flush0, flush1* and *flush2*, in instQ/RF, RF/EXE, EXE/MEM pipe latches, respectively. There are precedence in flush 0, 1 and 2, i.e. when *flush1* is asserted, *flush0* is also asserted. When a flush is asserted (synchronous), the pipeline latch will generate a nop in the next clock cycle, ignoring the current instruction that was supposed be passed to the next stage.
- determines if this flush updates archPC with new PC value
- PC of the first instruction to be inserted after the flush, if updating archPC is needed.
- Note that the outgoing flush signal to the traslation part is *flush0*. Since *flush1* and *2* are ORed to *flush0*, any of flush conditions also assert *flush0*.

Table 12: flush_unit

Signal	Bits	In/out	Description	Connections
flush0	1	OUT	flushes instQ (and all translation pipeline)	if a later-stage flush is asserted, earlier-stage flush signals are also asserted
flush1	1	OUT	flushes RF/EXE pipe latch	
flush2	1	OUT	flushes EXE/MEM pipe latch	
targetPC	16	OUT	targetPC of branch/jump/paging (PC of the first inst after the paging)	to PC_update/targetPC
PCupdate	1	OUT	asserted if nextPC should be reflected to archPC (branch/jump or paging)	to PC_update/PCupdate
br_targetPC	16	IN	targetPC from branch_unit	branch_unit/targetPC
br_taken	1	IN	taken/untaken from branch_unit	branch_unit/taken
paging_RQ	1	IN	signals that there is a (pending) paging request	from paging_detect/paging
interrupt_RQ	1	IN	signals that there is a (pending) interrupt request	from interrupt handler
mem_pipe_stall	1	IN	memory stall condition	from memory stage
bubble	1	IN	mem-to-alu dependence condition	from dep_check logic
EOI_RF_stage	1	IN	EOI bit of the instruction in RF stage	
EOI_EXE_stage	1	IN	EOI bit of the instruction in EXE stage	
EOI_MEM_stage	1	IN	EOI bit of the instruction in MEM stage	
PC_EXE_stage	16	IN	PC (next seqPC) of the instruction in EXE stage	
PC_MEM_stage	16	IN	PC (next seqPC) of the instruction in MEM stage	

Table 13: flush_unit description

```
flush_unit()
{
  // default output
  flush2 = 0;
  flush1 = 0;
  flush0 = 0;
  targetPC = 0;
  PCupdate = 0;

  if (mem_pipe_stall) return;

  // check paging
  if(paging)
  {
    // try to find nearest EOI
    if(EOI_MEM_stage) // find one in mem. ok to
flush2
    {
      flush2 = 1;
      flush1 = 1;
      flush0 = 1;
      targetPC = PC_MEM_stage;
      PCupdate = 1;
    }
    else if(EOI_EXE_stage || br_taken) // find one
in exe or taken branch
    {
      flush2 = 0;
      flush1 = 1;
      flush0 = 1;
      targetPC = br_taken ? br_targetPC :
PC_EXE_stage;
      PCupdate = 1;
    }
    else if(EOI_RF_stage) // find one in RF
    {
      if(bubble)
      {
        flush2 = 0;
        flush1 = 0;
        flush0 = 0;
        targetPC = 0;
        PCupdate = 0;
      }
      else
      {
        flush2 = 0;
        flush1 = 0;
        flush0 = 1;
        targetPC = 0;
        PCupdate = 0;
      }
    }
    else // no EOI, no taken branch. wait
    {
      flush2 = 0;
      flush1 = 0;
      flush0 = 0;
      targetPC = 0;
      PCupdate = 0;
    }
  }  // end of paging case
  else
  if (br_taken)
  {
    flush2 = 0;
    flush1 = 1;
    flush0 = 1;
    targetPC = br_targetPC;
    PCupdate = 1;
  } // end of branch taken case
  else
  if (interrupt_RQ) // interrupt request
  {
    // since br_taken case has already been han-
dled,
    // we deal with only EOI in RF stage
    if(EOI_RF_stage) // EOI in RF
    {
      if(bubble)
      {
        flush2 = 0;
        flush1 = 0;
        flush0 = 0;
        targetPC = 0;
        PCupdate = 0;
      }
      else
      {
        flush2 = 0;
        flush1 = 0;
        flush0 = 1;
        targetPC = 0;
        PCupdate = 0;
      }
    }
    else // no EOI in RF stage. wait
    {
      flush2 = 0;
      flush1 = 0;
      flush0 = 0;
      targetPC = 0;
      PCupdate = 0;
    }
  } // end of interrupt request
  else
  {
    flush2 = 0;
    flush1 = 0;
    flush0 = 0;
    targetPC = 0;
    PCupdate = 0;
  }
}
```

2.3.7 PC update unit

PC update unit is responsible for maintaining the correct archPC state. archPC is updated by PC_disp field in the instruction word. Usually, the first instruction in the translation group has a non-zero PC_disp that updates the current archPC to the next sequential PC value. This unit contains archPC, a special purpose register inside.

Table 14: PC_update

Signal	Bits	In/out	Description	Connections
CLK	1	IN	CLK	

Table 14: PC_update

Signal	Bits	In/out	Description	Connections
RST	1	IN	reset archPC to 0	
PC_disp	3	IN	PC displacement. addeded to the curr PC (archPC). Since these 3 bits are UNSIGNED, it should be 0 extended to 16 bits before it goes to adder	
archPC	16	OUT	the current content of archPC. archPC <= next_archPC every clk cycle	from the output of archPC (register)
next_archPC	16	OUT	if (pipe_stall) next_archPC <= archPC elseif(PCupdate) next_archPC <= targetPC else next_archPC <= sequential PC from the adder	to the input of archPC (register)
targetPC	16	IN	targetPC of branch/jump/paging	from flush_unit/nex-tPC
PCupdate	1	IN	if true (and !pipe_stall), n_archPC <= nextPC	from flush_unit/PCup-date
pipe_stall	1	IN	pipe stall condition?	(= mem_pipe_stall | bubble)

2.3.8 Interrupt handler

This unit samples INT/NMI signal from outside every posedge CLK, checks the current state of IFF1/IFF2 to determine if interrupt requests should be serviced. If ((INT && IFF1) | NMI), the unit sends an interrupt_RQ to flush_unit, and holds it until flush0 is asserted before the interrupt_RQ is reset. intern_INT and intern_NMI to the translation part are generated in the same clock cycle when flush0 occurs so that flush0 / intern_INT / intern_NMI are latched simultaneously at posedge CLK.

This unit contains IFF1 and IFF2, and supports IFF manipulations

Table 15: interrupt_handler

Signal	Bits	In/out	Description	Connections
CLK	1	IN	CLK	
RST	1	IN	resets interrupt_RQ, IFF1, IFF2	
extern_INT	1	IN	external INT	z80 definition
extern_NMI	1	IN	external NMI	z80 definition
intern_INT	1	OUT	internal INT. generated in sync with flush0	to xlator/INT
intern_NMI	1	OUT	internal NMI. generated in sync with flush0	to xlator/NMI
interrupt_RQ	1	OUT	set when ((INT && IFF1) | NMI). reset when flush0 occurs	to flush_unit/interrupt_RQ
flush0	1	IN	sees if the interrupt can be handled.	
IFF1	1	OUT	current content of IFF1	
IFF2	1	OUT	current content of IFF2	
EI	1	IN	enable interrupt (set IFF)	asserted by EI instruction in EXE stage
DI	1	IN	disable interrupt (reset IFF)	asserted by DI instruc-tion in EXE stage

2.3.9 paging handler

It monitors if memory is being written at 0xFFF? (should find some info -- ikim) and asserts paging_RQ to

flush_unit in the same clock cycle. paging_RQ is reset when flush0 occurs.

Table 16: paging_handler

Signal	Bits	In/out	Description	Connections
CLK	1	IN	CLK	
RST	1	IN	resets paging_RQ	
MREQ	1	IN	access request to memory	
addr	16	IN	16-bit memory address	
Rd_Wr	1	IN	checks if the request is read / write	
paging_RQ	1	OUT	paging request	to flush_unit
flush0	1	IN	reset paging_RQ if this signal comes in	

2.3.10 dependence check

It checks if any source operands of an instruction in RF stage are dependent on a load target operand in EXE stage. When detected, it asserts *bubble* to create a bubble into pipelatch1 between RF and EXE stages, in order to cover the incomplete bypass and avoid structural hazards.

Table 17: dep_check

Signal	Bits	In/out	Description	Connections
EXE_opcode	6	IN	opcode of the instruction in EXE stage to see it is load / IN instruction.	
EXE_Wr_id	5	IN	destination register id of the inst in EXE stage	
EXE_Fmask	8	IN	Fmask of the instruction in EXE stage	
Rd0_id	5	IN	src register id of the inst in RF stage	
Rd1_id	5	IN	src register id of the inst in RF stage	
bubble	1	OUT	asserts if a dependence is detected	

Note that *bubble* affects pipelaatch1 only when mem_pipe_stall does not occur. *bubble* is basically identical to *flush1* that puts a nop w/ !EOI.

2.3.11 misc signals

Table 18: misc signals

Signal	Bits	In/out	Description	Connections
pipe_stall	1		= mem_pipe_stall | bubble	from core to instruction queue
mem_pipe_stall	1		= (MREQ | IORQ)& D_wait	