1 Processor Design & Implementation

1.1 Y86-64 ISA

Table 1: Y86-64 instructions

Instruction	0		1		2 - 8	9
halt	0	0				
nop	1	0				
rrmovq rA rB	2	0	rA	rB		
irmovq V rB	3	0	F	rB	V	
rmmovq rA D(rB)	4	0	rA	rB	D	
mrmovq D(rB) rA	5	0	rA	rB	D	
OPq rA rB	6	fn	rA	rB		
jXX Dest	7	fn	Dest			
cmovXX rA rB	2	fn	rA	rB		
call Dest	8	0	Dest			
ret	9	0				
pushq rA	A	0	rA	F		
popq rA	B	0	rA	F		

Table 2: Function codes of Y86-64 instructions

addq	6	0	subq	6	1	andq	6	2	xorq	6	3
jmp	7	0	jle	7	1	jl	7	2	je	7	3
jne	7	4	jge	7	5	jg	7	6			
rrmovq	2	0	cmovle	2	1	cmovl	2	2	cmove	2	3
cmovne	2	4	cmovge	2	5	cmovg	7	6			

Table 3: Registers in Y86-64 ISA

%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
0	1	2	3	4	5	6	7
%r8	%r9	%r10	%r11	%r12	%r13	%r14	None
8	9	A	B	C	D	E	F

1.2 SEQ implementation of Y86-64

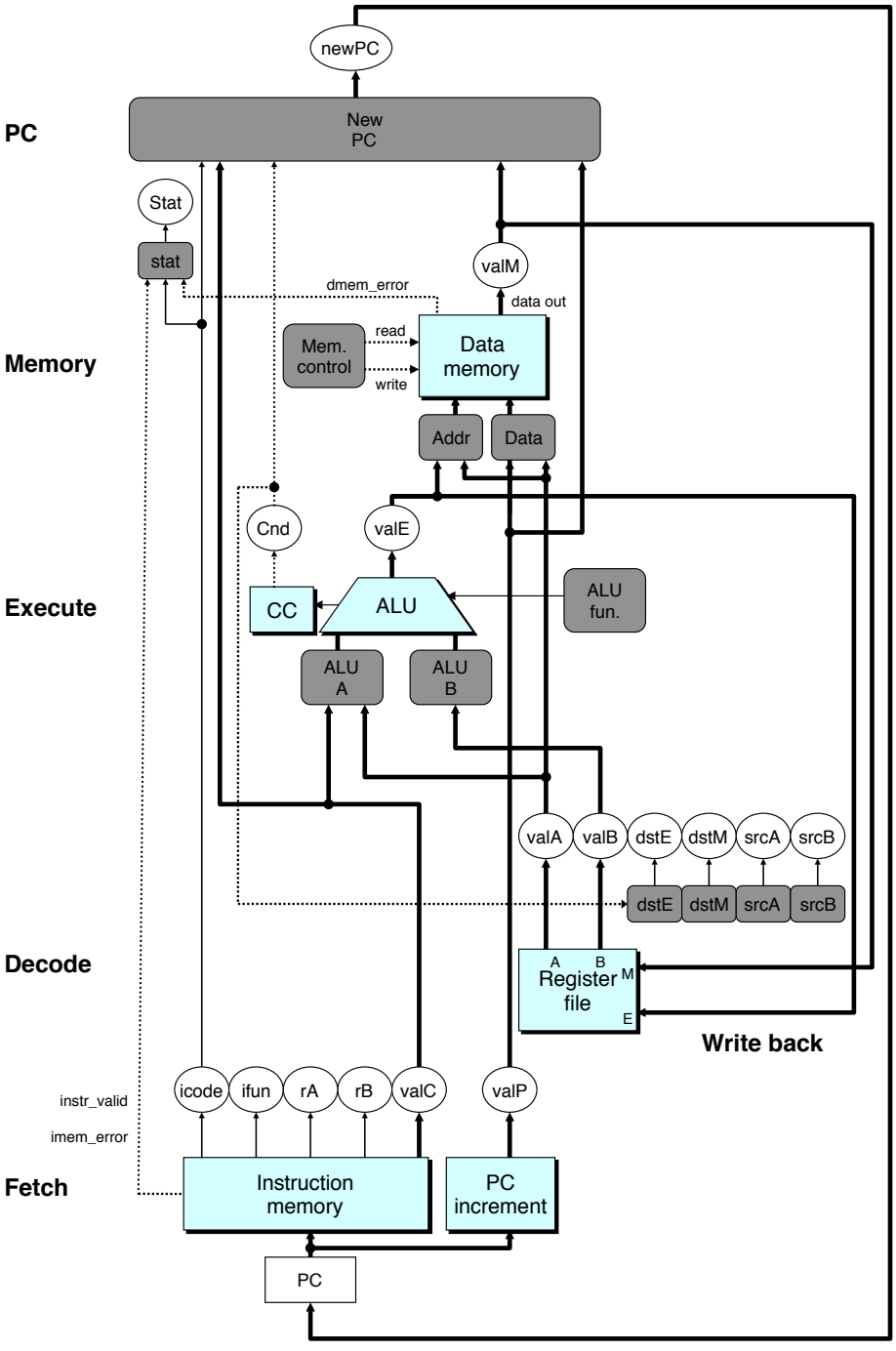

Figure 1: Sequential implementation of Y86-64

Table 4: Sequential implementation of Y86-64

Stage	Fetch	Decode	Execute	Memory	Write Back	PC Update
OPq rA rB	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[rA] valB ← R[rB]	valE ← valB OP valA Set CC		R[rB] ← valE	PC ← valP
rrmovq rA rB	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[rA]	valE ← 0+valA		R[rB] ← valE	PC ← valP
irmovq V rB	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10		valE ← 0+valC		R[rB] ← valE	PC ← valP
rmmovq rA D(rB)	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	valA ← R[rA] valB ← R[rB]	valE ← valB+valC	M_8[valE] ← valA		PC ← valP
mrmovq D(rB) rA	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	valB ← R[rB]	valE ← valB+valC	valM ← M_8[valE]	R[rA] ← valM	PC ← valP
pushq rA	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[rA] valB ← R[%rsp]	valE ← valB+(-8)	M_8[valE] ← valA	R[%rsp] ← valE	PC ← valP
popq rA	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[%rsp] valB ← R[%rsp]	valE ← valB+8	valM ← M_8[valA]	R[%rsp] ← valE R[rA] ← valM	PC ← valP
jXX Dest	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9		Cnd ← Cond(CC, ifun)			PC ← Cnd ? valC : valP
cmoveXX rA rB	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[rA]	valE ← 0+valA Cnd ← Cond(CC, ifun)		if(Cnd) R[rB] ← valE	PC ← valP
call Dest	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	valB ← R[%rsp]	valE ← valB+(-8)	M_8[valE] ← valP	R[%rsp] ← valE	PC ← valC
ret	icode:ifun ← M_1[PC] valP ← PC+1	valA ← R[%rsp] valB ← R[%rsp]	valE ← valB+8	valM ← M_8[valA]	R[%rsp] ← valE	PC ← valM

The following design is taken from both the book and the web aside.

- Fetch

```
word icode = [
  imem_error : INOP;
  1 : imem_icode;
];
word ifun = [
  imem_error : FNONE;
  1 : imem_ifun;
];
bool instr_valid = icode in {IHALT, INOP, IIRMOVQ, IMRMOVQ, IRMMOVQ, IRRMOVQ,
    ICALL, IRET, IPUSHQ, IPOPQ, IOPQ, IJXX};
bool need_regids = icode in {IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIRMOVQ, IRMMOVQ,
    IMRMOVQ};
bool need_valC = icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ, ICALL, IJXX};
```

  valP = PC + 1 + need_regids + 8 * need_valC.

- Decode & Write Back

```
word srcA = [
  icode in {IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ} : rA;
  icode in {IPOPQ, IRET} : RRSP;
  1 : RNONE;
];
word srcB = [
  icode in {IOPQ, IRMMOVQ, IMRMOVQ} : rB;
  icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP;
  1 : RNONE;
];
word dstE = [
  icode in {IOPQ, IIRMOVQ} : rB;
  icode == IRRMOVQ && Cnd : rB;
  icode in {ICALL, IRET, IPUSHQ, IPOPQ} : RRSP;
  1 : RNONE;
];
word dstM = [
  icode = {IPOPQ, IMRMOVQ} : rA;
  1 : RNONE;
];
```

- Execute

```
word aluA = [
  icode in {IOPQ, IRRMOVQ} : valA;
  icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ} : valC;
  icode in {IPUSHQ, ICALL} : -8;
  icode in {IPOPQ, IRET} : 8;
];
word aluB = [
  icode in {IOPQ, IRMMOVQ, IMRMOVQ, IPUSHQ, IPOPQ, ICALL, IRET} : valB;
  icode in {IIRMOVQ, IRRMOVQ} : 0;
];
word alufun = [
  icode == IOPQ: ifun;
  1 : ALUADD;
```

```
];
bool set_cc = icode in {IOPQ};
```

- Memory

```
bool mem_read = icode in {IMRMOVQ, IPOPQ, IRET};
bool mem_write = icode in {IRMMOVQ, IPUSH, ICALL};
word mem_addr = [
  icode in {IRMMOVQ, IMRMOVQ, IPUSHQ, ICALL} : valE;
  icode in {IPOPQ, IRET} : valA;
];
word mem_data = [
  icode in {IPUSHQ, IRMMOVQ} : valA;
  icode == ICALL : valP;
];
word Stat = [
  imem_error || dmem_error : SADR;
  !instr_valid : SINS;
  icode == IHLT : SHLT;
  1 : SAOK;
];
```

- PC Update

```
word new_pc = [
  icode == ICALL : valC;
  icode == IJXX && Cnd : valC;
  icode == IRET : valM;
  1 : valP;
];
```

1.3 Pipeline implementation of Y86-64

- Fetch & PC Selection

```
word f_pc = [
  M_icode == IJXX && !M_Cnd : M_valA; //When jump PC prediction is wrong
  W_icode == IRET : W_valM;
  1 : F_predPC;
];
word f_predPC = [
  f_icode in {IJXX, ICALL} : f_valC; //Strategy for jump PC prediction: always
      jump
  1 : f_valP;
];
word f_stat = [
  imem_error : SADR; // dmem_error in memory phase
  !instr_valid : SINS;
  f_icode == IHLT : SHLT;
  1 : SAOK;
];

// similar to SEQ
word f_icode = [
  imem_error : INOP;
  1 : imem_icode;
];
```

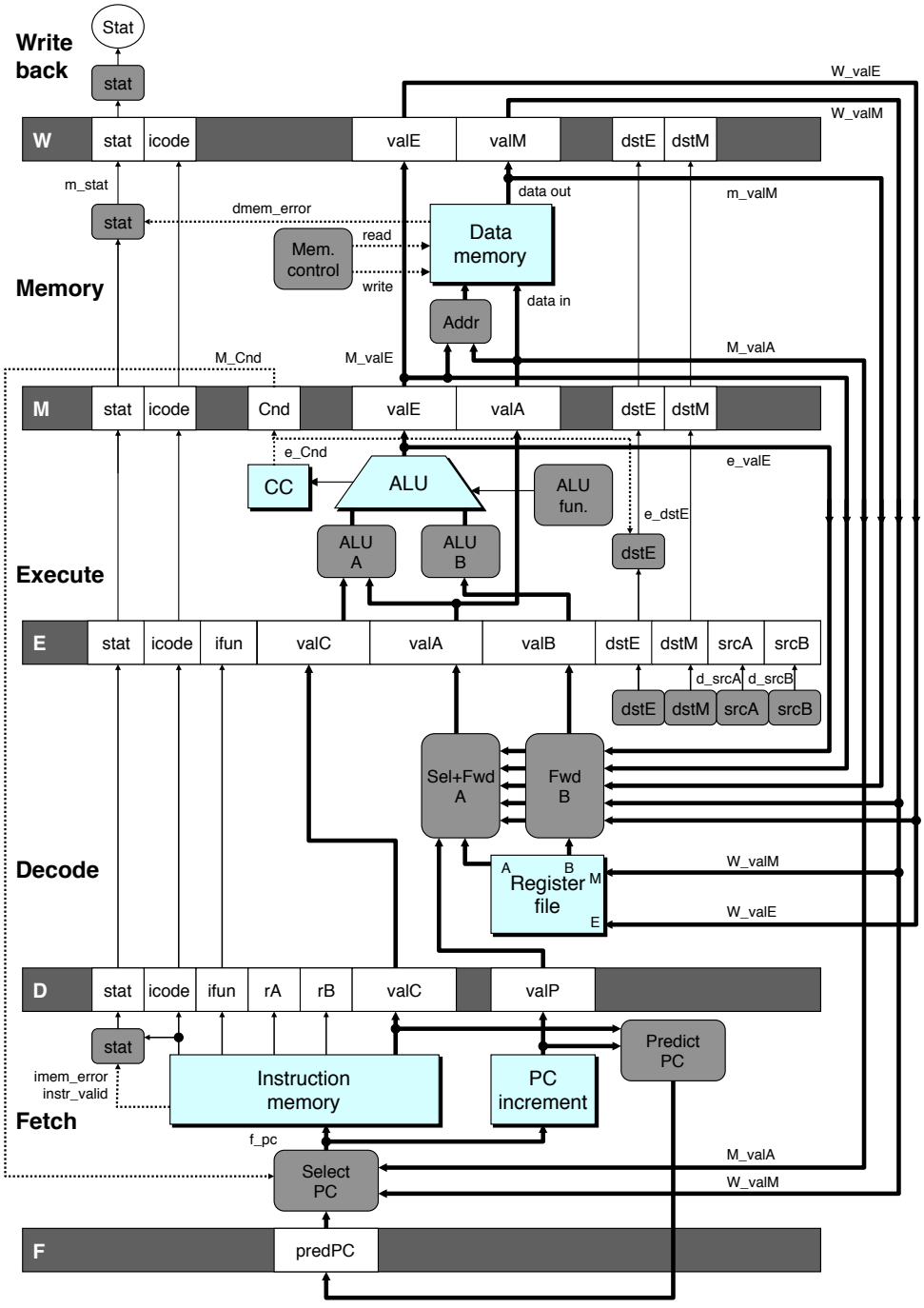

Figure 2: Pipeline implementation of Y86-64

```
word f_ifun = [
  imem_error : FNONE;
  1 : imem_ifun;
];
bool instr_valid = f_icode in {IHALT, INOP, IIRMOVQ, IMRMOVQ, IRMMOVQ, IRRMOVQ,
     ICALL, IRET, IPUSHQ, IPOPQ, IOPQ, IJXX};
bool need_regids = f_icode in {IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIRMOVQ, IRMMOVQ,
    IMRMOVQ};
bool need_valC = f_icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ, ICALL, IJXX};
```

- Decode & Write Back

  Precedence of forwarding sources:

  - Forward data affected by the nearest instruction. Thus E > M > W.
  - For correct behavior of `popq %rsp`, M > E.

```
word d_valA = [
  D_icode in {ICALL, IJXX} : D_valP;
  d_srcA == e_dstE : e_valE;
  d_srcA == M_dstM : m_valM;
  d_srcA == M_dstE : M_valE;
  d_srcA == W_dstM : W_valM;
  d_srcA == W_dstE : W_valE;
  1 : d_rvalA;
];
word d_valB = [
  d_srcB == e_dstE : e_valE;
  d_srcB == M_dstM : m_valM;
  d_srcB == M_dstE : M_valE;
  d_srcB == W_dstM : W_valM;
  d_srcB == W_dstE : W_valE;
  1 : d_rvalB;
];
word w_dstE = W_dstE;
word w_dstM = W_dstM;
word w_valE = W_valE;
word w_valM = W_valM;
word Stat = [
  W_stat = SBUB : SAOK;
  1 : W_stat;
];
// similar to SEQ
word d_srcA = [
  D_icode in {IOPQ, IRMMOVQ, IRRMOVQ, IPUSHQ} : D_rA;
  D_icode in {IRET, IPOPQ} : RRSP;
  1 : RNONE;
];
word d_srcB = [
  D_icode in {IOPQ, IRMMOVQ, IMRMOVQ} : D_rB;
  D_icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP;
  1 : RNONE;
];
word d_dstE = [
  D_icode in {IOPQ, IIRMOVQ} : rB;
  D_icode in {ICALL, IRET, IPUSHQ, IPOPQ} : RRSP;
  1 : RNONE;
```

```
];
word d_dstM = [
  D_icode = {IPOPQ, IMRMOVQ} : D_rA;
  1 : RNONE;
];
```

- Execute

```
bool set_cc = E_icode == IOPQ && !m_stat in {SADR, SHLT, SINS} && !W_stat in {
    SADR, SHLT, SINS};
word e_valA = E_valA;
word e_dstE = [
  E_icode == IRRMOVQ && !e_Cnd : RNONE;
  1 : E_dstE;
];
// similar to SEQ
word aluA = [
  E_icode in {IOPQ, IRRMOVQ} : E_valA;
  E_icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ} : E_valC;
  E_icode in {IPUSHQ, ICALL} : -8;
  E_icode in {IPOPQ, IRET} : 8;
];
word aluB = [
  E_icode in {IOPQ, IRMMOVQ, IMRMOVQ, IPUSHQ, IPOPQ, ICALL, IRET} : E_valB;
  E_icode in {IIRMOVQ, IRRMOVQ} : 0;
];
word alufun = [
  E_icode == IOPQ: E_ifun;
  1 : ALUADD;
];
```

- Memory

```
word m_stat = [
  dmem_error : SADR;
  1 : M_stat;
];
// similar to SEQ
bool mem_read = M_icode in {IMRMOVQ, IPOPQ, IRET};
bool mem_write = M_icode in {IRMMOVQ, IPUSH, ICALL};
word mem_addr = [
  M_icode in {IRMMOVQ, IMRMOVQ, IPUSHQ, ICALL} : M_valE;
  M_icode in {IPOPQ, IRET} : M_valA;
];
```

1.4 Special pipeline control logic

```
bool F_bubble = 0;
bool F_stall = E_code in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB} ||
  IRET in {D_icode, E_icode, M_icode};
bool D_stall = E_code in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB} ||;
bool D_bubble = E_code == IJXX && !e_Cnd ||
  IRET in {D_icode, E_icode, M_icode} && !(E_code in {IMRMOVQ, IPOPQ} && E_dstM in {
      d_srcA, d_srcB});
bool E_stall = 0;
bool E_bubble = E_code == IJXX && !e_Cnd ||
```

Table 5: Special pipeline control logic

	Condition	F	D	E	M	W
ret	`IRET ∈ {D_icode, E_icode, M_icode}`	stall	bubble	normal	normal	normal
load / use hazard	`E_icode ∈ {IMRMOVQ, IPOPQ} && E_dstM ∈ {d_srsA, d_srcB}`	stall	stall	bubble	normal	normal
mispredicted branch	`E_icode == IJXX && !e_Cnd`	normal	bubble	bubble	normal	normal
exception	`m_stat ∈ {SADR, SHLT, SINS} || W_stat ∈ {SADR, SHLT, SINS}`	normal	normal	normal	bubble	stall (only W)
ret & misprediction	jumps to `ret`	stall	bubble	bubble	normal	normal
ret & load / use	set `%rsp` followed by `ret`	stall	stall	bubble	normal	normal

```
  E_code in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB};
bool M_bubble = m_stat in {SADR, SHLT, SINS} ||
  W_stat in {SADR, SHLT, SINS};
bool M_stall = 0;
bool W_bubble = 0;
//Do not stall when m_stat is not OK. Otherwise the instruction in W (which does not
    cause exception) won't finish.
bool W_stall = W_stat in {SADR, SINS, SHLT};
```