**Questions:**

Exception has a higher priority than interrupts

Check for higher priority for exception

Do we store the sum for the add.asm file as a word or as a byte

./simulator ucode4 add.obj data.obj except\_prot.obj except\_unaligned.obj int.obj except\_unknown.obj vector\_table.obj

Page 713 of the textbook is where al lthe graph stuff is

Page 343 is interrupts

Check at the end of the cycle that there are no exceptions – if there are that’s when you know what state to go to

mportant note: we should be able to swap out your exception routines with a routine that returns from an exception. Therefore, make sure you implement the exceptions correctly.)

dumpsim for protection exception

Memory content [0x4000..0x4002] :

-------------------------------------

0x4000 (16384) : 0x0001

0x4002 (16386) : 0x0000

Current register/bus values :

-------------------------------------

Cycle Count : 299

PC : 0x301a

IR : 0x1442

STATE\_NUMBER : 0x0021

BUS : 0x0000

MDR : 0x0000

MAR : 0x3018

CCs: N = 0 Z = 0 P = 1

Registers:

0: 0xc003

1: 0x0023

2: 0x007f

3: 0x0011

4: 0x0000

5: 0x0000

6: 0x0000

7: 0x0000

Memory content [0x4000..0x4004] :

-------------------------------------

0x4000 (16384) : 0x0002

0x4002 (16386) : 0x0000

0x4004 (16388) : 0x0000

Current register/bus values :

-------------------------------------

Cycle Count : 750

PC : 0x301c

IR : 0x8000

STATE\_NUMBER : 0x0021

BUS : 0x301a

MDR : 0x8004

MAR : 0x301a

CCs: N = 1 Z = 0 P = 0

Registers:

0: 0xc004

1: 0x0023

2: 0x007f

3: 0x0011

4: 0x0000

5: 0x0000

6: 0x0000

7: 0x0000

Memory content [0x4000..0x4004] :

-------------------------------------

0x4000 (16384) : 0x0002

0x4002 (16386) : 0x0000

0x4004 (16388) : 0x0000

Current register/bus values :

-------------------------------------

Cycle Count : 1708

PC : 0x0000

IR : 0xf025

STATE\_NUMBER : 0x0012

BUS : 0x0000

MDR : 0x0000

MAR : 0x004a

CCs: N = 0 Z = 1 P = 0

Registers:

0: 0x0000

1: 0x0007

2: 0x0052

3: 0x0000

4: 0x0000

5: 0x0000

6: 0x2ffc

7: 0x1602

**Rdump and mdump for unaligned**

Memory content [0x4000..0x4004] :

-------------------------------------

0x4000 (16384) : 0x0002

0x4002 (16386) : 0x0000

0x4004 (16388) : 0x0000

Current register/bus values :

-------------------------------------

Cycle Count : 1708

PC : 0x0000

IR : 0xf025

STATE\_NUMBER : 0x0012

BUS : 0x0000

MDR : 0x0000

MAR : 0x004a

CCs: N = 1 Z = 0 P = 0

Registers:

0: 0xc017

1: 0x0007

2: 0x0052

3: 0x0000

4: 0x0000

5: 0x0000

6: 0x2ffc

7: 0x1a02

CODE :

#define INSTR(pos, wid,instr) ((instr) >> (pos) & ~(~0<<(wid)))

#define Low8bits(x) ((x) & 0x00FF)

#define High8bits(x) ((x) & 0xFF00)

#define SIGN\_EXT(num,wid) ((num) >> (wid) - 1? ~0 <<(wid) | (num) :(num))

#define ZERO\_EXTEND(num,pos,wid) ((num) >> (pos) & ~(~0 << (wid)))

#define Table 0x0200

int inter\_request = 0;

int value\_gateSP=0;

int value\_gatePSR=0;

int value\_gateTable=0;

int first\_cycle = 0;

int mem\_cycle=0;

int value\_gatePC=0;

int value\_gateMDR=0;

int value\_gateALU=0;

int value\_gateMARMUX=0;

int value\_gateSHF=0;

int main\_addr\_output=0;

int value\_gatePCFinal=0;

void eval\_micro\_sequencer() {

/\*

\* Evaluate the address of the next state according to the

\* micro sequencer logic. Latch the next microinstruction.

\*/

if(first\_cycle==0){

CURRENT\_LATCHES.PSR = 0x8000;

first\_cycle=1;

}

if(CYCLE\_COUNT==300){

inter\_request=1;

}

int state = CURRENT\_LATCHES.STATE\_NUMBER;

int IRD = GetIRD(CURRENT\_LATCHES.MICROINSTRUCTION);

int j = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

int cond = GetCOND(CURRENT\_LATCHES.MICROINSTRUCTION);

int add\_mode = INSTR(11,1,CURRENT\_LATCHES.IR);

int next\_instr[CONTROL\_STORE\_BITS];

printf("\nBEGINNING OF NEW STATE\n\nSTATE NUMBER RIGHT NOW: %d\n\n",CURRENT\_LATCHES.STATE\_NUMBER);

printf("\nCYCLE COUNT: %d\n\n",CYCLE\_COUNT);

if(IRD==1){

int next\_state = INSTR(12,4,CURRENT\_LATCHES.IR);

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[next\_state][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = next\_state;

return;

}

else if(IRD==0){

if(cond==0){

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

if(cond==1){

if(CURRENT\_LATCHES.READY==1){

j+=2;

NEXT\_LATCHES.READY=0;

}

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

if(cond==2){

if(CURRENT\_LATCHES.BEN==1){

j+=4;

}

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

if(cond==3){

if(add\_mode==1){

j+=1;

}

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

if(cond==4){

int jsr15 = INSTR(15,1,CURRENT\_LATCHES.PSR);

if(jsr15==1){

j+=8;

}

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

}

}

void cycle\_memory() {

/\*

\* This function emulates memory and the WE logic.

\* Keep track of which cycle of MEMEN we are dealing with.

\* If fourth, we need to latch Ready bit at the end of

\* cycle to prepare microsequencer for the fifth cycle.

\*/

int mem\_use = GetMIO\_EN(CURRENT\_LATCHES.MICROINSTRUCTION);

if(mem\_use==1){

mem\_cycle++;

}

else{

return;

}

}

void eval\_bus\_drivers() {

/\*

\* Datapath routine emulating operations before driving the bus.

\* Evaluate the input of tristate drivers

\* Gate\_MARMUX,

\* Gate\_PC,

\* Gate\_ALU,

\* Gate\_SHF,

\* Gate\_MDR.

\*/

int spMux = GetSPMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int psrMux = GetPSRMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int psrMux\_output = 0;

int data\_size = GetDATA\_SIZE(CURRENT\_LATCHES.MICROINSTRUCTION);

int updated\_mar = CURRENT\_LATCHES.MAR >> 1;

if(data\_size==0){

int mar\_zero = INSTR(0,1,CURRENT\_LATCHES.MAR);

if(mar\_zero==0){

value\_gateMDR=Low16bits(SIGN\_EXT((MEMORY[updated\_mar][0]),8));

}

if(mar\_zero==1){

value\_gateMDR = Low16bits(SIGN\_EXT((MEMORY[updated\_mar][1]),8));

}

}

if(data\_size==1){

value\_gateMDR = CURRENT\_LATCHES.MDR;

}

int dr\_mux = GetDRMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int immm5 = INSTR(0,5,CURRENT\_LATCHES.IR);

int am4 = INSTR(0,4,CURRENT\_LATCHES.IR);

int off6 = INSTR(0,6,CURRENT\_LATCHES.IR);

int off9 = INSTR(0,9,CURRENT\_LATCHES.IR);

int off11 = INSTR(0,11,CURRENT\_LATCHES.IR);

int sr2mux\_output;

int bit5= INSTR(5,1,CURRENT\_LATCHES.IR);

if(bit5==0){

sr2mux\_output = CURRENT\_LATCHES.REGS[INSTR(0,3,CURRENT\_LATCHES.IR)];

}

else if(bit5==1){

sr2mux\_output = SIGN\_EXT(immm5,5);

}

int sr1mux = GetSR1MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int addr1mux = GetADDR1MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int addr2mux = GetADDR2MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int marmux = GetMARMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int aluk = GetALUK(CURRENT\_LATCHES.MICROINSTRUCTION);

int pcmux = GetPCMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int lshf = GetLSHF1(CURRENT\_LATCHES.MICROINSTRUCTION);

int sr1mux\_output;

int dr1mux\_output;

int addr1mux\_output;

int addr2mux\_output;

if(sr1mux==0){

int reg\_num = INSTR(9,3,CURRENT\_LATCHES.IR);

sr1mux\_output = CURRENT\_LATCHES.REGS[reg\_num];

}

else if(sr1mux==1){

int reg\_num = INSTR(6,3,CURRENT\_LATCHES.IR);

sr1mux\_output = CURRENT\_LATCHES.REGS[reg\_num];

}

else if(sr1mux==2){

sr1mux\_output = CURRENT\_LATCHES.REGS[6];

}

if(dr\_mux==0){

int reg\_num = INSTR(9,3,CURRENT\_LATCHES.IR);

dr1mux\_output = CURRENT\_LATCHES.REGS[reg\_num];

}

else if(dr\_mux==1){

dr1mux\_output = CURRENT\_LATCHES.REGS[7];

}

else if(dr\_mux==2){

dr1mux\_output = CURRENT\_LATCHES.REGS[6];

}

if(addr1mux==0){

addr1mux\_output= CURRENT\_LATCHES.PC;

}

else if(addr1mux==1){

addr1mux\_output = sr1mux\_output;

}

//sp mux is after this

if(spMux==0){

value\_gateSP= CURRENT\_LATCHES.SSP;

}

if(spMux==1){

value\_gateSP= sr1mux\_output-2;

}

if(spMux==2){

value\_gateSP= sr1mux\_output+2;

}

if(spMux==3){

value\_gateSP= CURRENT\_LATCHES.USP;

}

if(addr2mux==0){

addr2mux\_output=0;

}

if(addr2mux==1){

addr2mux\_output= SIGN\_EXT(off6,6);

}

if(addr2mux==2){

addr2mux\_output= SIGN\_EXT(off9,9);

}

if(addr2mux==3){

addr2mux\_output= SIGN\_EXT(off11,11);

}

if(lshf==1){

addr2mux\_output = addr2mux\_output<<1;

}

main\_addr\_output= addr2mux\_output+addr1mux\_output;

if(aluk==0){

value\_gateALU = sr2mux\_output+sr1mux\_output;

}

else if(aluk==1){

value\_gateALU = sr2mux\_output & sr1mux\_output;

}

else if(aluk==2){

value\_gateALU = sr2mux\_output ^ sr1mux\_output;

}

else if(aluk==3){

if(data\_size==1){

value\_gateALU = sr1mux\_output;

}

else if(data\_size==0){

value\_gateALU = (Low8bits(sr1mux\_output)<<8)+(Low8bits(sr1mux\_output));

}

}

int shift= INSTR(4, 2,CURRENT\_LATCHES.IR);

if(shift==0){

value\_gateSHF= sr1mux\_output<<am4;

}

else if(shift==1){

value\_gateSHF = sr1mux\_output>>am4;

}

else if(shift==3){

value\_gateSHF = (SIGN\_EXT(sr1mux\_output, 16) >> am4);

}

if(marmux==0){

int ir7 = INSTR(0,8,CURRENT\_LATCHES.IR);

value\_gateMARMUX= (ZERO\_EXTEND(ir7,0,8))<<1;

}

else if(marmux==1){

value\_gateMARMUX = main\_addr\_output;

}

value\_gatePCFinal=CURRENT\_LATCHES.PC;

if(pcmux==0){

value\_gatePC = CURRENT\_LATCHES.PC;

}

else if(pcmux==1){

value\_gatePC = BUS;

}

else if(pcmux==2){

value\_gatePC = main\_addr\_output;

}

else if(pcmux==3){

value\_gatePCFinal=CURRENT\_LATCHES.PC-2;

}

value\_gatePSR = CURRENT\_LATCHES.PSR;

value\_gateTable = CURRENT\_LATCHES.VECT + Table;

}

void drive\_bus() {

/\*

\* Datapath routine for driving the bus from one of the 5 possible

\* tristate drivers.

\*/

int GateTable = GetGATE\_TABLE(CURRENT\_LATCHES.MICROINSTRUCTION);

int GatePSR = GetGATE\_PSR(CURRENT\_LATCHES.MICROINSTRUCTION);

int GateSP = GetGATE\_SP(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gatepc = GetGATE\_PC(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gatemdr = GetGATE\_MDR(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gatealu = GetGATE\_ALU(CURRENT\_LATCHES.MICROINSTRUCTION);

int GateMarmux = GetGATE\_MARMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gateshf = GetGATE\_SHF(CURRENT\_LATCHES.MICROINSTRUCTION);

if(Gatepc || Gatemdr || Gatealu || GateMarmux || Gateshf || GateSP || GatePSR || GateTable) {

if(GateTable){

BUS = Low16bits(value\_gateTable);

}

if(GateSP) {

BUS = Low16bits(value\_gateSP);

}

if(GatePSR) {

BUS = Low16bits(value\_gatePSR);

}

if (Gatepc) {

BUS = Low16bits(value\_gatePCFinal);

}

if (Gatemdr) {

BUS = Low16bits(value\_gateMDR);

}

if (Gatealu) {

BUS = Low16bits(value\_gateALU);

}

if (GateMarmux) {

BUS = Low16bits(value\_gateMARMUX);

}

if (Gateshf) {

BUS = Low16bits(value\_gateSHF);

}

}

else{

BUS= Low16bits(0);

}

printf("Gate Table: %X\n Gate Sp: %x\nGate PSR: %X\n Gate PC: %x\nGate MDR: %X\n Gate ALU: %x\nGate MARmux: %X\n Gate shf: %x\n",value\_gateTable,value\_gateSP,value\_gatePSR,value\_gatePCFinal,value\_gateMDR,value\_gateALU,value\_gateMARMUX,value\_gateSHF);

}

void latch\_datapath\_values() {

/\*

\* Datapath routine for computing all functions that need to latch

\* values in the data path at the end of this cycle. Some values

\* require sourcing the bus; therefore, this routine has to come

\* after drive\_bus.

\*/

// do psr mux and latching for psr here

int PSR\_changed = 0;

int ld\_psr = GetLD\_PSR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_exc = GetLD\_EXEC(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_ssp = GetLD\_SSP(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_usp = GetLD\_USP(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_vector = GetLD\_VECTOR(CURRENT\_LATCHES.MICROINSTRUCTION);

printf("\n\nLD values: PSR: %d\n exc:%d\nSSP: %d\n USP:%d\nvector: %d\n\n",ld\_psr,ld\_exc,ld\_ssp,ld\_usp,ld\_vector);

if(mem\_cycle==MEM\_CYCLES-1){

NEXT\_LATCHES.READY=1;

}

int ld\_mar = GetLD\_MAR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_mdr = GetLD\_MDR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_ir = GetLD\_IR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_ben = GetLD\_BEN(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_reg = GetLD\_REG(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_cc = GetLD\_CC(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_pc = GetLD\_PC(CURRENT\_LATCHES.MICROINSTRUCTION);

int dr\_mux = GetDRMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

if(ld\_psr){

int psrMux = GetPSRMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

if(psrMux==0){

NEXT\_LATCHES.PSR = CURRENT\_LATCHES.PSR & 0x7FFF;

PSR\_changed=1;

}

if(psrMux==1){

NEXT\_LATCHES.PSR = BUS;

int n = INSTR(2,1,NEXT\_LATCHES.PSR);

int z = INSTR(1,1,NEXT\_LATCHES.PSR);

int p = INSTR(0,1,NEXT\_LATCHES.PSR);

PSR\_changed=1;

if(z){

NEXT\_LATCHES.Z=1;

NEXT\_LATCHES.P=0;

NEXT\_LATCHES.N=0;

}

else if(n){

NEXT\_LATCHES.Z=0;

NEXT\_LATCHES.P=0;

NEXT\_LATCHES.N=1;

}

else if(p){

NEXT\_LATCHES.Z=0;

NEXT\_LATCHES.P=1;

NEXT\_LATCHES.N=0;

}

}

}

if(ld\_ssp){

int sr1Mux = GetSR1MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int sr = INSTR(9,3,CURRENT\_LATCHES.IR);

if(sr1Mux==1){

sr = INSTR(6,3,CURRENT\_LATCHES.IR);

}

if(sr1Mux==2){

sr=6;

}

NEXT\_LATCHES.SSP = CURRENT\_LATCHES.REGS[sr];

printf("Loading ssp in - register: %d\t and ssp for register: %X\n\n",sr,CURRENT\_LATCHES.REGS[sr]);

}

if(ld\_usp){

int sr1Mux = GetSR1MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int sr = INSTR(9,3,CURRENT\_LATCHES.IR);

if(sr1Mux==1){

sr = 7;

}

if(sr1Mux==2){

sr=6;

}

NEXT\_LATCHES.USP = CURRENT\_LATCHES.REGS[sr];

}

if(ld\_vector){

NEXT\_LATCHES.VECT = CURRENT\_LATCHES.VECT<<1;

}

if(ld\_mar){

NEXT\_LATCHES.MAR = BUS;

}

if(ld\_mdr){

NEXT\_LATCHES.MDR = BUS;

}

if(ld\_ir){

NEXT\_LATCHES.IR = BUS;

}

if(ld\_ben){

int n = CURRENT\_LATCHES.N;

int z = CURRENT\_LATCHES.Z;

int p = CURRENT\_LATCHES.P;

int ir\_n = INSTR(11,1,CURRENT\_LATCHES.IR);

int ir\_z = INSTR(10,1,CURRENT\_LATCHES.IR);

int ir\_p = INSTR(9,1,CURRENT\_LATCHES.IR);

NEXT\_LATCHES.BEN = (n & ir\_n) + (z & ir\_z) + (p & ir\_p);

}

if(ld\_reg){

int dr = INSTR(9,3,CURRENT\_LATCHES.IR);

if(dr\_mux==1){

dr=7;

}

if(dr\_mux==2){

dr=6;

printf("right register");

}

NEXT\_LATCHES.REGS[dr]= BUS;

}

if(ld\_cc){

int sign\_bit= INSTR(15,1,BUS);

if(BUS==0){

NEXT\_LATCHES.Z=1;

NEXT\_LATCHES.P=0;

NEXT\_LATCHES.N=0;

}

else if(sign\_bit){

NEXT\_LATCHES.Z=0;

NEXT\_LATCHES.P=0;

NEXT\_LATCHES.N=1;

}

else if(sign\_bit==0){

NEXT\_LATCHES.Z=0;

NEXT\_LATCHES.P=1;

NEXT\_LATCHES.N=0;

}

}

if(ld\_pc){

int pc\_mux = GetPCMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

if(pc\_mux==0){

NEXT\_LATCHES.PC = CURRENT\_LATCHES.PC+2;

}

else if(pc\_mux==1){

NEXT\_LATCHES.PC = BUS;

}

else if(pc\_mux==2){

NEXT\_LATCHES.PC = main\_addr\_output;

}

}

if(mem\_cycle==MEM\_CYCLES) {

mem\_cycle=0;

int state = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

int read\_write = GetR\_W(CURRENT\_LATCHES.MICROINSTRUCTION);

int data\_size = GetDATA\_SIZE(CURRENT\_LATCHES.MICROINSTRUCTION);

int updated\_mar = CURRENT\_LATCHES.MAR >> 1;

int current\_mdr = CURRENT\_LATCHES.MDR;

if (read\_write) {

if (data\_size) {

//state 16

int msb\_md = (current\_mdr & 0xFF00)>>8;

int lsb\_md = current\_mdr & 0x00FF;

MEMORY[updated\_mar][0] = lsb\_md;

MEMORY[updated\_mar][1] = msb\_md;

} else {

//state 17

int mar\_zero = INSTR(0, 1, CURRENT\_LATCHES.MAR);

MEMORY[updated\_mar][mar\_zero] = Low8bits(current\_mdr);

}

} else {

if (ld\_mdr==1) {

//state 25,28,29,33

NEXT\_LATCHES.MDR = Low16bits(

(Low8bits(MEMORY[updated\_mar][0])) + High8bits((MEMORY[updated\_mar][1]) << 8));

}

}

}

int interr = inter\_request && CURRENT\_LATCHES.STATE\_NUMBER==18;

int exception\_caused = 0;

int next\_instr[CONTROL\_STORE\_BITS];

int next\_MAR\_value = NEXT\_LATCHES.MAR;

int IRD = GetIRD(CURRENT\_LATCHES.MICROINSTRUCTION);

int J = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

int psr15 = INSTR(15,1,CURRENT\_LATCHES.PSR);

printf("right before the next exception 3 opcode - this is the state number: %d", NEXT\_LATCHES.STATE\_NUMBER);

if(NEXT\_LATCHES.STATE\_NUMBER==10 || NEXT\_LATCHES.STATE\_NUMBER==11){

printf("\n\nexception 3 opcode\n");

exception\_caused=1;

NEXT\_LATCHES.VECT = opcode;

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[54][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = 54;

}

if(IRD==2){

if(0<=next\_MAR\_value && next\_MAR\_value<=0x2FFF && psr15==1){

printf("\n\nexception 1 protection\n\n");

exception\_caused=1;

NEXT\_LATCHES.VECT = protect;

}

int unaligned\_check = INSTR(0,1,NEXT\_LATCHES.MAR);

printf("unaligned value: %d",unaligned\_check);

int next\_j = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[next\_j][i];

}

int testing\_off = 0;

if(GetDATA\_SIZE(next\_instr) && unaligned\_check && exception\_caused==0){

exception\_caused=1;

printf("\n\nexception 2 unaligned\n\n");

NEXT\_LATCHES.VECT = unaligned;

}

if(interr==1 && exception\_caused==0){

printf("\n\ninterruption\n Cycle Count: %d\n\n",CYCLE\_COUNT);

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[38][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = 38;

NEXT\_LATCHES.VECT = interruptLC<<1;

inter\_request=0;

}

if(exception\_caused==1){

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[54][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = 54;

}

if(exception\_caused==0 && interr==0){

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[J][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = J;

}

}

if(PSR\_changed==0){

if(NEXT\_LATCHES.Z==1){

NEXT\_LATCHES.PSR = (CURRENT\_LATCHES.PSR | 0x0002) & 0xFFF2;

}

if(NEXT\_LATCHES.P==1){

NEXT\_LATCHES.PSR = (CURRENT\_LATCHES.PSR | 0x0001) & 0xFFF1;

}

if(NEXT\_LATCHES.N==1){

NEXT\_LATCHES.PSR = (CURRENT\_LATCHES.PSR | 0x0004) & 0xFFF4;

}

}

printf("\n\nCurrent Latches: \nVECT: %X\nSSp: %X\nUSP: %X\nPSR: %X\n",CURRENT\_LATCHES.VECT,CURRENT\_LATCHES.SSP,CURRENT\_LATCHES.USP,CURRENT\_LATCHES.PSR);

printf("\n\nNext Latches: \nVECT: %X\nSSp: %X\nUSP: %X\nPSR: %X\n\n",NEXT\_LATCHES.VECT,NEXT\_LATCHES.SSP,NEXT\_LATCHES.USP,NEXT\_LATCHES.PSR);

printf("\nEND of state\n\nNEXTSTATE NUMBER: %d\n\n",NEXT\_LATCHES.STATE\_NUMBER);

}