Final Project – Progress Report 3

CS 385 – Computer Architecture

Daniel Kostecki, Thi Nguyen, and Sonia Leonato Soiras

**Tasks:**

Daniel Kostecki:

* Gate level conversions in the regfile
* Regfile implementation in project
* Report and adjusted diagrams
* Code comments
* Testing program

Thi Nguyen:

* 16-bit implementation of multiplexors using 4x1 mux
* Regfile D-flip-flops
* CPU module
* Main Control Unit
* I-type operations for Report 2
* Testing program

Sonia Leonato Soiras:

* ALU conversions to 16-bit
* Gate level conversions in the ALU (4x1 mux and 2x1 mux)
* Truth table
* Code debugging
* Testing program

**Descriptions:**

The instruction set architecture currently in place for our 16-bit, 3-stage pipelined machine allows for R-type and addi instructions. The list of R-type instructions that are currently supported and their respective opcode are and(0000), sub(0001), add(0010), or(0011), and slt(0111). The opcodes of the I-type instructions are addi(0100).

R-type format: I-type format:

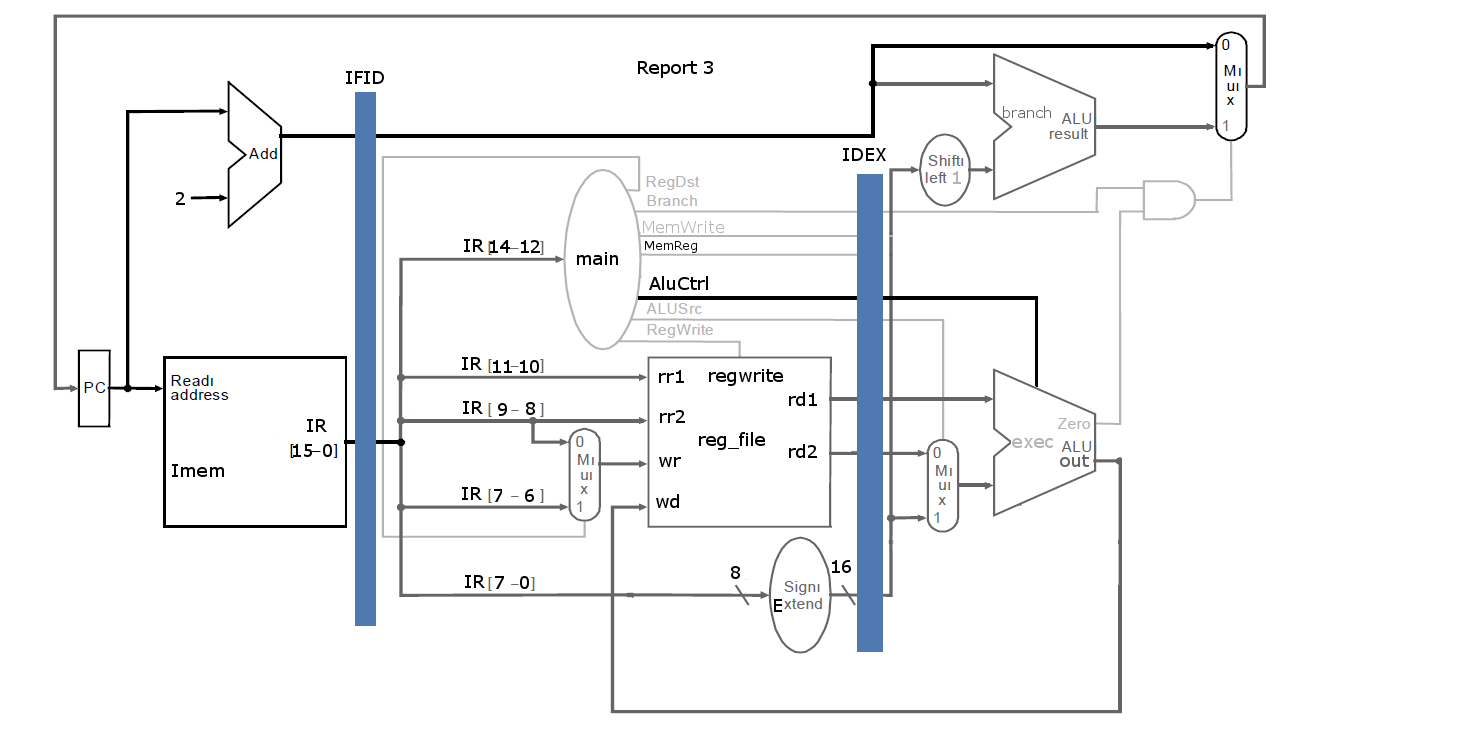
|  |  |  |  |
| --- | --- | --- | --- |
| Op | Rs | Rt | Address/Value |
| 4 | 2 | 2 | 8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op | Rs | Rt | Rd | Unused |
| 4 | 2 | 2 | 2 | 6 |

The format of the instructions is dependent on the type if instruction. For example, in the r-type format we first get the op code (listed above). This is followed by the two registers which are being used. The final register is the destination. Because we have a 4-bit opcode followed by 3 2-bit registers this leaves us with 6-bits at the end of every r-type that is unused. For the i-type format this changes. The op code is still 4-bit, and there are still two registers (rs being used and rt being destination) but this is followed by an 8-bit value or address.

**Logic Diagrams/Truth Tables:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Instr** | **RegDst** | **ALUSrc** | **MemReg** | **RegWrite** | **MemWrite** | **Branch** | **ALUCtrl** |
| **add** | 1 | 0 | 0 | 1 | 0 | 00 | 010 |
| **sub** | 1 | 0 | 0 | 1 | 0 | 00 | 110 |
| **and** | 1 | 0 | 0 | 1 | 0 | 00 | 000 |
| **or** | 1 | 0 | 0 | 1 | 0 | 00 | 001 |
| **slt** | 1 | 0 | 0 | 1 | 0 | 00 | 111 |
| **addi** | 0 | 1 | 0 | 1 | 0 | 00 | 010 |

****

**Verilog Code:**

/\*

By: Daniel Kostecki, Sonia Leonato, Thi Nguyen

Class: CS 385

Progress Report 3 - Implementing a 16-bit MIPS machine in Verilog

\*/

/\*

16-bit register file with input of read register 1(rr1),

read register 2 (rr2), write register (wr), write data (wd),

regwrite, and clock. The outputs are read data 1 (rd1),

and read data 2 (rd2). The register file relies on registers,

a 16 bit mux, and a decoder.

\*/

module reg\_file (rr1,rr2,wr,wd,regwrite,rd1,rd2,clock);

input [1:0] rr1,rr2,wr;

input [15:0] wd;

input regwrite,clock;

output [15:0] rd1,rd2;

wire [15:0] q1, q2, q3;

//registers

register r1 (wd,c1,q1),

r2 (wd,c2,q2),

r3 (wd,c3,q3);

//output port

mux4x1\_16bit mux1 (16'b0,q1,q2,q3,rr1,rd1),

mux2 (16'b0,q1,q2,q3,rr2,rd2);

//input port

decoder dec (wr[1],wr[0],w3,w2,w1,w0);

and a (regwrite\_and\_clock,regwrite,clock);

and a1 (c1,regwrite\_and\_clock,w1),

a2 (c2,regwrite\_and\_clock,w2),

a3 (c3,regwrite\_and\_clock,w3);

endmodule

// The register module is implemented using 16 D-flip-flops.

module register(D,CLK,Q);

input [15:0]D;

input CLK;

output [15:0]Q;

D\_flip\_flop d1 (D[0],CLK,Q[0]),

d2 (D[1],CLK,Q[1]),

d3 (D[2],CLK,Q[2]),

d4 (D[3],CLK,Q[3]),

d5 (D[4],CLK,Q[4]),

d6 (D[5],CLK,Q[5]),

d7 (D[6],CLK,Q[6]),

d8 (D[7],CLK,Q[7]),

d9 (D[8],CLK,Q[8]),

d10(D[9],CLK,Q[9]),

d11(D[10],CLK,Q[10]),

d12(D[11],CLK,Q[11]),

d13(D[12],CLK,Q[12]),

d14(D[13],CLK,Q[13]),

d15(D[14],CLK,Q[14]),

d16(D[15],CLK,Q[15]);

endmodule

// The D-flip-flop module implemented using d-latches.

module D\_flip\_flop (D,CLK,Q);

input D,CLK;

output Q;

wire CLK1, Y;

not not1 (CLK1,CLK);

D\_latch D1(D,CLK, Y),

D2(Y,CLK1,Q);

endmodule

// The d-latch module.

module D\_latch (D,C,Q);

input D,C;

output Q;

wire x,y,D1,Q1;

nand nand1 (x,D, C),

nand2 (y,D1,C),

nand3 (Q,x,Q1),

nand4 (Q1,y,Q);

not not1 (D1,D);

endmodule

/\*

The 16-bit ALU which takes an operation (op) input as well as

an 'a' and 'b' input. What is returned is reult which is sent

to the write data (wd) of the register file, and zero.

\*/

module ALU (op,a,b,result,zero);

input [15:0] a;

input [15:0] b;

input [2:0] op;

output [15:0] result;

output zero;

wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;

ALU1 alu0 (a[0],b[0],op[2],op[1:0],set,op[2],c1,result[0]),

alu1 (a[1],b[1],op[2],op[1:0],0,c1,c2,result[1]),

alu2 (a[2],b[2],op[2],op[1:0],0,c2,c3,result[2]),

alu3 (a[3],b[3],op[2],op[1:0],0,c3,c4,result[3]),

alu4 (a[4],b[4],op[2],op[1:0],0,c4,c5,result[4]),

alu5 (a[5],b[5],op[2],op[1:0],0,c5,c6,result[5]),

alu6 (a[6],b[6],op[2],op[1:0],0,c6,c7,result[6]),

alu7 (a[7],b[7],op[2],op[1:0],0,c7,c8,result[7]),

alu8 (a[8],b[8],op[2],op[1:0],0,c8,c9,result[8]),

alu9 (a[9],b[9],op[2],op[1:0],0,c9,c10,result[9]),

alu10(a[10],b[10],op[2],op[1:0],0,c10,c11,result[10]),

alu11(a[11],b[11],op[2],op[1:0],0,c11,c12,result[11]),

alu12(a[12],b[12],op[2],op[1:0],0,c12,c13,result[12]),

alu13(a[13],b[13],op[2],op[1:0],0,c13,c14,result[13]),

alu14(a[14],b[14],op[2],op[1:0],0,c14,c15,result[14]);

ALUmsb alu15(a[15],b[15],op[2],op[1:0],0,c15,c16,result[15],set);

// If result = 0 then zero = 1

check\_zero cz(zero,result);

endmodule

/\*

1-bit ALU used to handle bits 0-14 (least significant bits) using a

2x1 and 4x1 multiplexor as well as a full adder.

\*/

module ALU1 (a,b,binvert,op,less,carryin,carryout,result);

input a,b,less,carryin,binvert;

input [1:0] op;

output carryout,result;

wire sum, a\_and\_b, a\_or\_b, b\_inv;

not not1(b\_inv, b);

mux2x1 mux1(b,b\_inv,binvert,b1);

and and1(a\_and\_b, a, b);

or or1(a\_or\_b, a, b);

fulladder adder1(sum,carryout,a,b1,carryin);

mux4x1 mux2(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);

endmodule

/\*

1-bit ALU used to handle bit 15 (the most significant bit)

using a 2x1 and 4x1 multiplexor as well as a full adder.

\*/

module ALUmsb (a,b,binvert,op,less,carryin,carryout,result,sum);

input a,b,less,carryin,binvert;

input [1:0] op;

output carryout,result,sum;

wire sum, a\_and\_b, a\_or\_b, b\_inv;

not not1(b\_inv, b);

mux2x1 mux1(b,b\_inv,binvert,b1);

and and1(a\_and\_b, a, b);

or or1(a\_or\_b, a, b);

fulladder adder1(sum,carryout,a,b1,carryin);

mux4x1 mux2(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);

endmodule

// Module to check zero returned by the ALU

module check\_zero (zero,result);

input [15:0] result;

wire z[14:0];

output zero;

or g5(z[0],result[0],result[1]);

or g6(z[1],result[2],result[3]);

or g7(z[2],result[4],result[5]);

or g8(z[3],result[6],result[7]);

or g9(z[4],result[8],result[9]);

or g1(z[5],result[10],result[11]);

or g2(z[6],result[12],result[13]);

or g3(z[7],result[14],result[15]);

or g4(z[8],z[0],z[1]);

or g10(z[9],z[2],z[3]);

or g11(z[10],z[4],z[5]);

or g12(z[11],z[6],z[7]);

or g13(z[12],z[8],z[9]);

or g14(z[13],z[10],z[11]);

or g15(z[14],z[12],z[13]);

xnor g8(zero,z[14],0);

endmodule

// Half adder module used twice in the implementation of the full adder.

module halfadder (S,C,x,y);

input x,y;

output S,C;

xor (S,x,y);

and (C,x,y);

endmodule

// Full adder module using two half adders.

module fulladder (S,C,x,y,z);

input x,y,z;

output S,C;

wire S1,D1,D2;

halfadder HA1 (S1,D1,x,y),

HA2 (S,D2,S1,z);

or g1(C,D2,D1);

endmodule

// 4x1 multiplexor module.

module mux4x1 (a\_and\_b, a\_or\_b,sum, less, op[1:0], result);

input a\_and\_b,a\_or\_b,sum,less;

input [1:0] op;

wire a,b,c,d,notOp0,notOp1;

output result;

not g1(notOp0,op[0]); // S0

not g2(notOp1,op[1]); // S1

and g3(a,a\_and\_b,notOp1,notOp0), // AND operation - 00

g4(b,a\_or\_b,notOp1,op[0]), // OR operation - 01

g5(c,sum,op[1],notOp0), // ADD operation - 10

g6(d,less,op[1],op[0]); // SLT operation - 11

or g7(result,a,b,c,d);

endmodule

// 16-bit 4x1 multiplexor module.

module mux4x1\_16bit (i0,i1,i2,i3,select,y);

input [15:0] i0,i1,i2,i3;

input [1:0] select;

output [15:0] y;

mux4x1 m1(i0[0], i1[0], i2[0], i3[0], select,y[0]);

mux4x1 m2(i0[1], i1[1], i2[1], i3[1], select,y[1]);

mux4x1 m3(i0[2], i1[2], i2[2], i3[2], select,y[2]);

mux4x1 m4(i0[3], i1[3], i2[3], i3[3], select,y[3]);

mux4x1 m5(i0[4], i1[4], i2[4], i3[4], select,y[4]);

mux4x1 m6(i0[5], i1[5], i2[5], i3[5], select,y[5]);

mux4x1 m7(i0[6], i1[6], i2[6], i3[6], select,y[6]);

mux4x1 m8(i0[7], i1[7], i2[7], i3[7], select,y[7]);

mux4x1 m9(i0[8], i1[8], i2[8], i3[8], select,y[8]);

mux4x1 m10(i0[9], i1[9], i2[9], i3[9], select,y[9]);

mux4x1 m11(i0[10],i1[10],i2[10],i3[10], select,y[10]);

mux4x1 m12(i0[11],i1[11],i2[11],i3[11],select,y[11]);

mux4x1 m13(i0[12],i1[12],i2[12],i3[12],select,y[12]);

mux4x1 m14(i0[13],i1[13],i2[13],i3[13],select,y[13]);

mux4x1 m15(i0[14],i1[14],i2[14],i3[14],select,y[14]);

mux4x1 m16(i0[15],i1[15],i2[15],i3[15],select,y[15]);

endmodule

// 2x1 multiplexor module.

module mux2x1 (A,B,select,OUT);

input A,B,select;

output OUT;

wire notS,andA,andB;

not g1(notS,select);

and g2(andA,A,notS),

g3(andB,B,select);

or g4(OUT,andA,andB);

endmodule

// 2-bit 2x1 multiplexor module.

module mux2x1\_2bit (A,B,select,y);

input [1:0] A,B;

input select;

output [1:0] y;

mux2x1 mux1(A[0], B[0], select, y[0]),

mux2(A[1], B[1], select, y[1]);

endmodule

// 16-bit 2x1 multiplexor module.

module mux2x1\_16bit (A, B, select, y);

input [15:0] A,B;

input select;

output [15:0] y;

mux2x1 mux1(A[0], B[0], select, y[0]),

mux2(A[1], B[1], select, y[1]),

mux3(A[2], B[2], select, y[2]),

mux4(A[3], B[3], select, y[3]),

mux5(A[4], B[4], select, y[4]),

mux6(A[5], B[5], select, y[5]),

mux7(A[6], B[6], select, y[6]),

mux8(A[7], B[7], select, y[7]),

mux9(A[8], B[8], select, y[8]),

mux10(A[9], B[9], select, y[9]),

mux11(A[10], B[10], select, y[10]),

mux12(A[11], B[11], select, y[11]),

mux13(A[12], B[12], select, y[12]),

mux14(A[13], B[13], select, y[13]),

mux15(A[14], B[14], select, y[14]),

mux16(A[15], B[15], select, y[15]);

endmodule

// Decoder module.

module decoder (S1,S0,D3,D2,D1,D0);

input S0,S1;

output D0,D1,D2,D3;

not n1 (notS0,S0),

n2 (notS1,S1);

and a0 (D0,notS1,notS0),

a1 (D1,notS1, S0),

a2 (D2, S1,notS0),

a3 (D3, S1, S0);

endmodule

// Branch Control

module BranchCtrl(Op, Zero, Out);

input [1:0] Op;

input Zero;

output Out;

wire pZero, i0,i1;

not n(pZero, Zero);

and a1(i0, Op[0], Zero),

a2(i1, Op[1], pZero);

or o(Out,i0,i1);

endmodule

// Main Control Unit module.

module MainControl (op,control);

input [3:0] op;

output reg [9:0] control;

//Control is in the format of: RegDst, AluSrc, MemtoReg, RegWrite, MemWrite, Branch, AluCtrl

always @(op) case (op)

//R-types

4'b0000: control <= 10'b1001000010; // ADD

4'b0001: control <= 10'b1001000110; // SUB

4'b0010: control <= 10'b1001000000; // AND

4'b0011: control <= 10'b1001000001; // OR

4'b0111: control <= 10'b1001000111; // SLT

//I-type

4'b0100: control <= 10'b0101000010; // ADDI

4'b0101: control <= 10'b0111000010; // LW

4'b0110: control <= 10'b0100100010; // SW

4'b1000: control <= 10'b0000001110; // BEQ

4'b1001: control <= 10'b0000010110; // BNE

endcase

endmodule

/\*

CPU module which implements the other major components (ALU, Register,

and control unit).

\*/

// #### cpu module remodled for report 3 ####

module CPU (clk, PC, IFID\_IR, IDEX\_IR, WD);

input clk;

output [15:0] PC, IFID\_IR, IDEX\_IR, WD;

reg[15:0] PC;

reg[15:0] Imem[0:1023];

wire [15:0] NextPC, AluOut, A, B, RD1, RD2, SignExtend;

reg [2:0] AluCtrl;

wire [1:0] WR;

//Test Program

initial begin

//r-type = op(4),rs(2), rt(2), rd(2), unused(6)

//i-type = op(4), rs(2), rt(2), address/value(8)

// Program with nop's - no hazards

Imem[0] = 16'b0100000100001111; // addi $t1, $0, 15 ($t1 = 15)

Imem[1] = 16'b0100001000000111; // addi $t2, $0, 7 ($t1 = 7)

Imem[2] = 16'b0000000000000000; // nop

Imem[3] = 16'b0010011011000000; // and $t3, $t1, $t2 ($t3 = 7)

Imem[4] = 16'b0000000000000000; // nop

Imem[5] = 16'b0001011110000000; // sub $t2, $t1, $t3 ($t2 = 8)

Imem[6] = 16'b0000000000000000; // nop

Imem[7] = 16'b0011101110000000; // or $t2, $t2, $t3 ($t2 = 15)

Imem[8] = 16'b0000000000000000; // nop

Imem[9] = 16'b0000101111000000; // add $t3, $t2, $t3 ($t3 = 22)

Imem[10]= 16'b0000000000000000; // nop

Imem[11]= 16'b0111111001000000; // slt $t1, $t3, $t2 ($t1 = 0)

Imem[12]= 16'b0111101101000000; // slt $t1, $t2, $t3 ($t1 = 1)

end

/\*

initial begin

// Program without nop's - wrong results

Imem[0] = 16'b0100000100001111; // addi $t1, $0, 15 ($t1 = 15)

Imem[1] = 16'b0100001000000111; // addi $t2, $0, 7 ($t1 = 7)

Imem[2] = 16'b0010011011000000; // and $t3, $t1, $t2 ($t3 = 7)

Imem[3] = 16'b0001011110000000; // sub $t2, $t1, $t3 ($t2 = 8)

Imem[4] = 16'b0011101110000000; // or $t2, $t2, $t3 ($t2 = 15)

Imem[5] = 16'b0000101111000000; // add $t3, $t2, $t3 ($t3 = 22)

Imem[6] = 16'b0111111001000000; // slt $t1, $t3, $t2 ($t1 = 0)

Imem[7] = 16'b0111101101000000; // slt $t1, $t2, $t3 ($t1 = 1)

end

\*/

// Pipeline stages

//=== IF STAGE ===

reg[15:0] IFID\_IR, IFID\_PC2;

//--------------------------------

ALU fetch(3'b010,PC,16'b10,NextPC,Unused);

//=== ID STAGE ===

wire [9:0] Control;

//----------------------------------------------------

reg [15:0] IDEX\_IR; // For monitoring the pipeline

reg IDEX\_RegWrite, IDEX\_ALUSrc, IDEX\_RegDst, IDEX\_MemToReg, IDEX\_MemWrite;

reg [1:0] IDEX\_Branch;

reg [15:0] IDEX\_RD1,IDEX\_RD2,IDEX\_SignExt;

reg [1:0] IDEX\_rt,IDEX\_rd;

//----------------------------------------------------

reg\_file rf (IFID\_IR[11:10],IFID\_IR[9:8],WR,WD,IDEX\_RegWrite,RD1,RD2,clk);

MainControl MainCtr (IFID\_IR[15:12],Control);

assign SignExtend = {{8{IFID\_IR[7]}},IFID\_IR[7:0]};

//=== EXE STAGE ===

ALU exec(AluCtrl, IDEX\_RD1, B, AluOut, Zero);

//assign B = (IDEX\_ALUSrc) ? IDEX\_SignExt: IDEX\_RD2; // ALUSrc Mux

mux2x1\_16bit muxB (IDEX\_RD2, IDEX\_SignExt, IDEX\_ALUSrc, B);

//assign WR = (IDEX\_RegDst) ? IDEX\_rd: IDEX\_rt; // RegDst Mux

mux2x1\_2bit muxWR (IDEX\_rt, IDEX\_rd, IDEX\_RegDst, WR);

assign WD = AluOut;

initial begin

PC = 0;

end

// Running the pipeline

always @(negedge clk) begin

// Stage 1 - IF

PC <= NextPC;

IFID\_IR <= Imem[PC>>1];

// Stage 2 - ID

IDEX\_IR <= IFID\_IR; // For monitoring the pipeline

{IDEX\_RegDst,IDEX\_ALUSrc, IDEX\_MemToReg, IDEX\_RegWrite, IDEX\_MemWrite, IDEX\_Branch, AluCtrl} <= Control;

IDEX\_RD1 <= RD1;

IDEX\_RD2 <= RD2;

IDEX\_SignExt <= SignExtend;

IDEX\_rt <= IFID\_IR[9:8];

IDEX\_rd <= IFID\_IR[7:6];

// Stage 3 - EX

// No transfers needed here - on negedge WD is written into register WR

end

endmodule

// Test module using the CPU.

module test ();

reg clock;

wire [15:0] PC,IFID\_IR,IDEX\_IR,WD;

CPU test\_cpu(clock,PC,IFID\_IR,IDEX\_IR,WD);

always #1 clock = ~clock;

initial begin

$display ("Time PC IFID\_IR IDEX\_IR WD");

$monitor ("%4d %3d %5h %8h %7d", $time,PC,IFID\_IR,IDEX\_IR,WD);

clock = 1;

#29 $finish;

end

endmodule

**Test results**

Time PC IFID\_IR IDEX\_IR WD

0 0 xxxx xxxx x

1 2 410f xxxx x

3 4 4207 410f 15

5 6 0000 4207 7

7 8 26c0 0000 0

9 10 0000 26c0 7

11 12 1780 0000 0

13 14 0000 1780 8

15 16 3b80 0000 0

17 18 0000 3b80 15

19 20 0bc0 0000 0

21 22 0000 0bc0 22

23 24 7e40 0000 0

25 26 7b40 7e40 0

27 28 xxxx 7b40 1

29 30 xxxx xxxx X