Zet x86 open source SoC

http://zet.aluzina.org

v1.1 19 Feb 2010

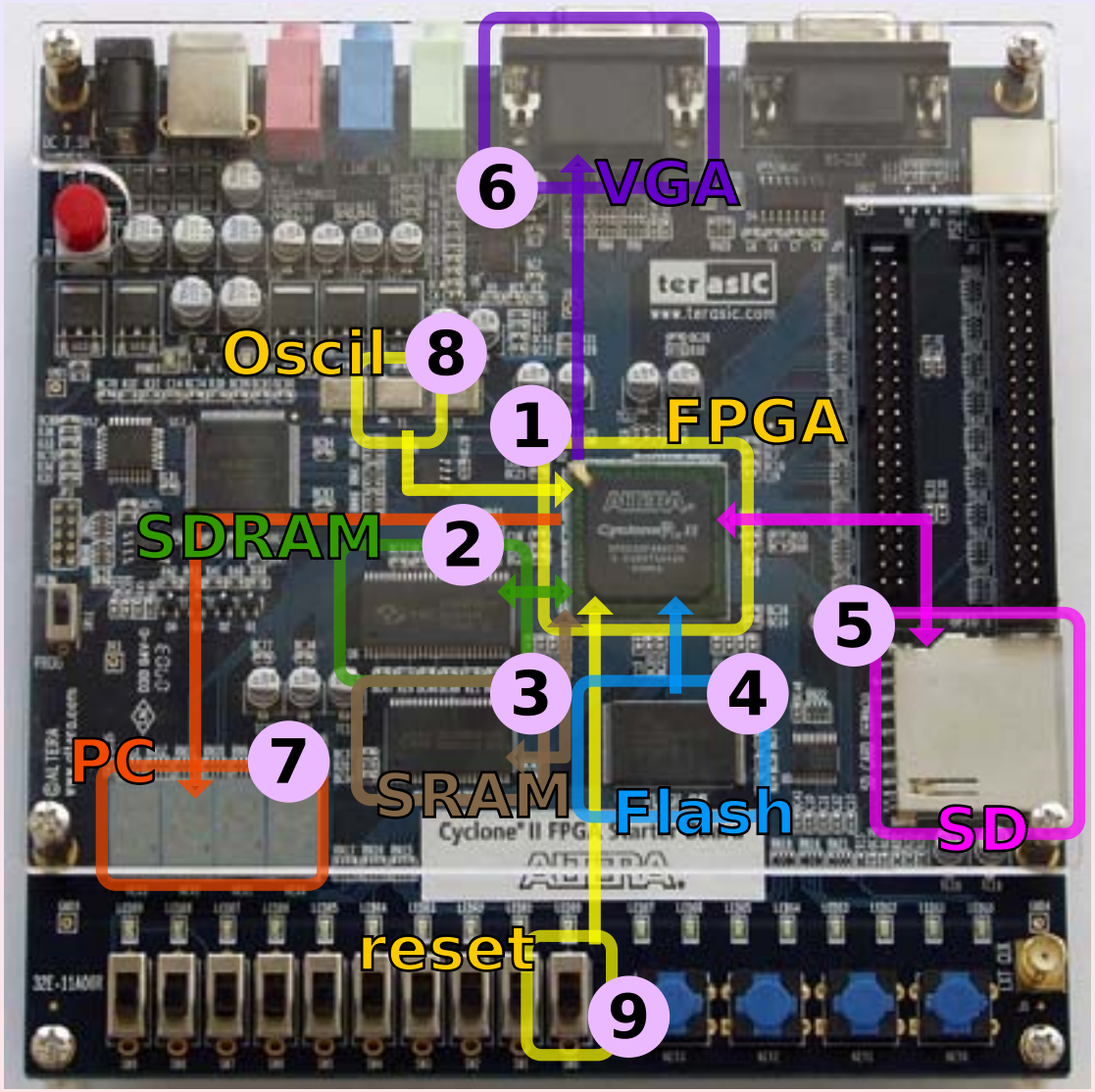
Zeus Gómez Marmolejo

Contents

Contents

Terasic Altera DE1 - Cyclone II FPGA

http://www.terasic.com.tw

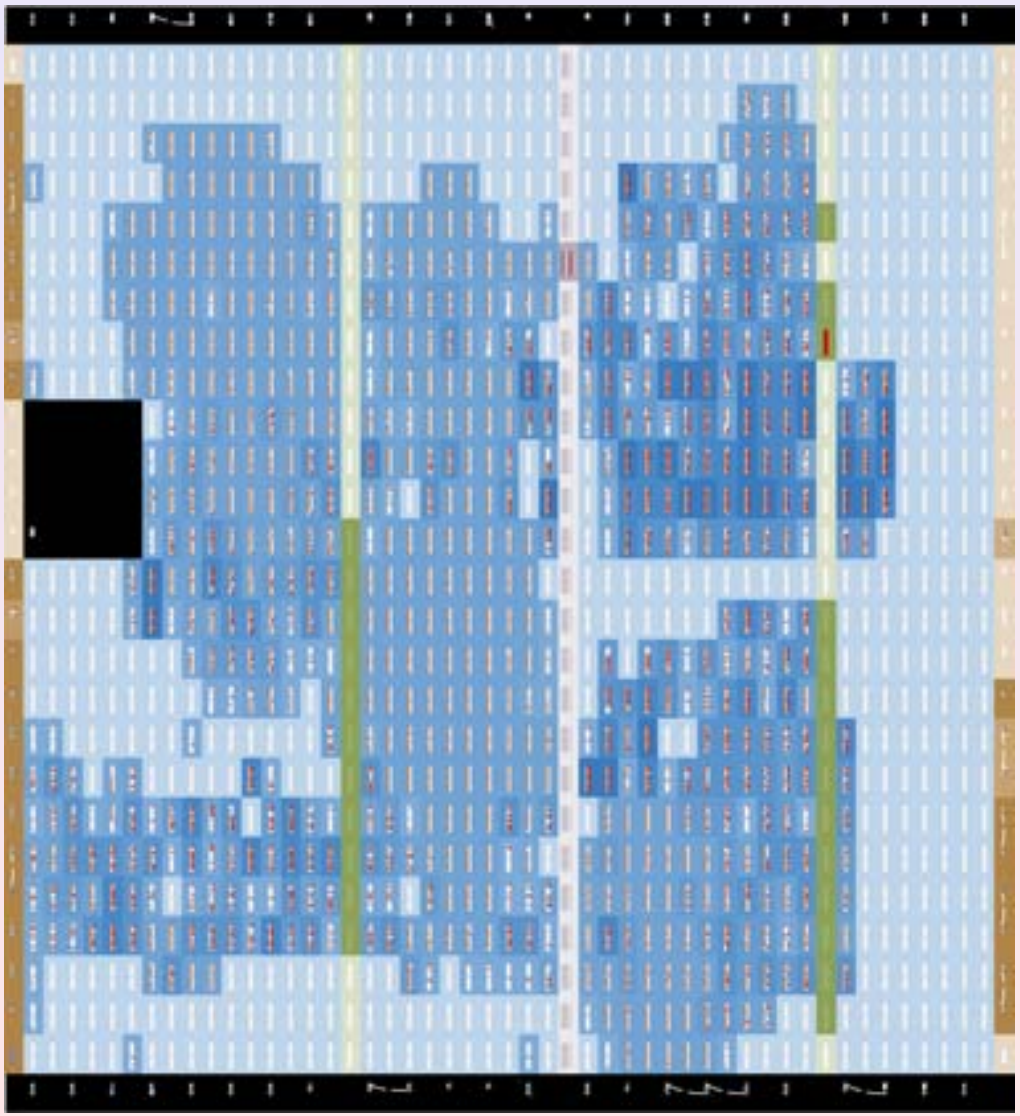


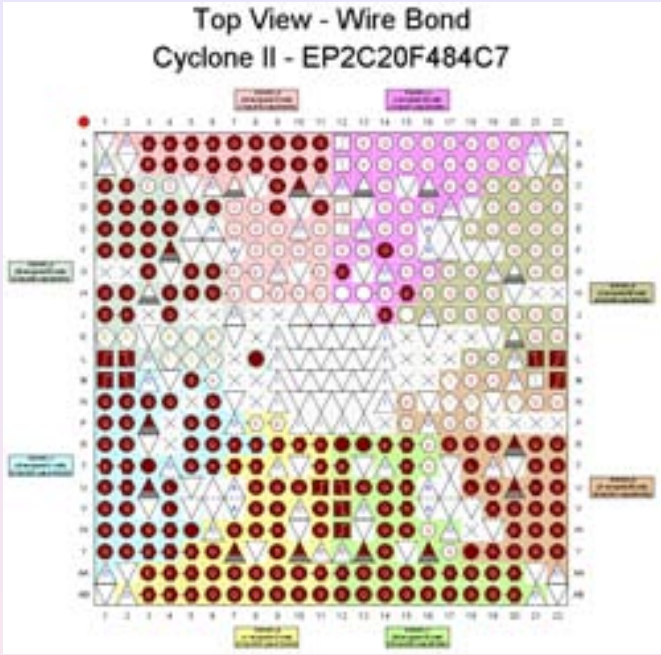
Physical devices

1. FPGA
2. SDRAM
3. SRAM
4. Flash
5. SD card
6. VGA
7. 50 Mhz osc
8. PC
9. reset

FPGA device EP2C20F484C7

Chip floorplan and pin usage for Zet version 1.1





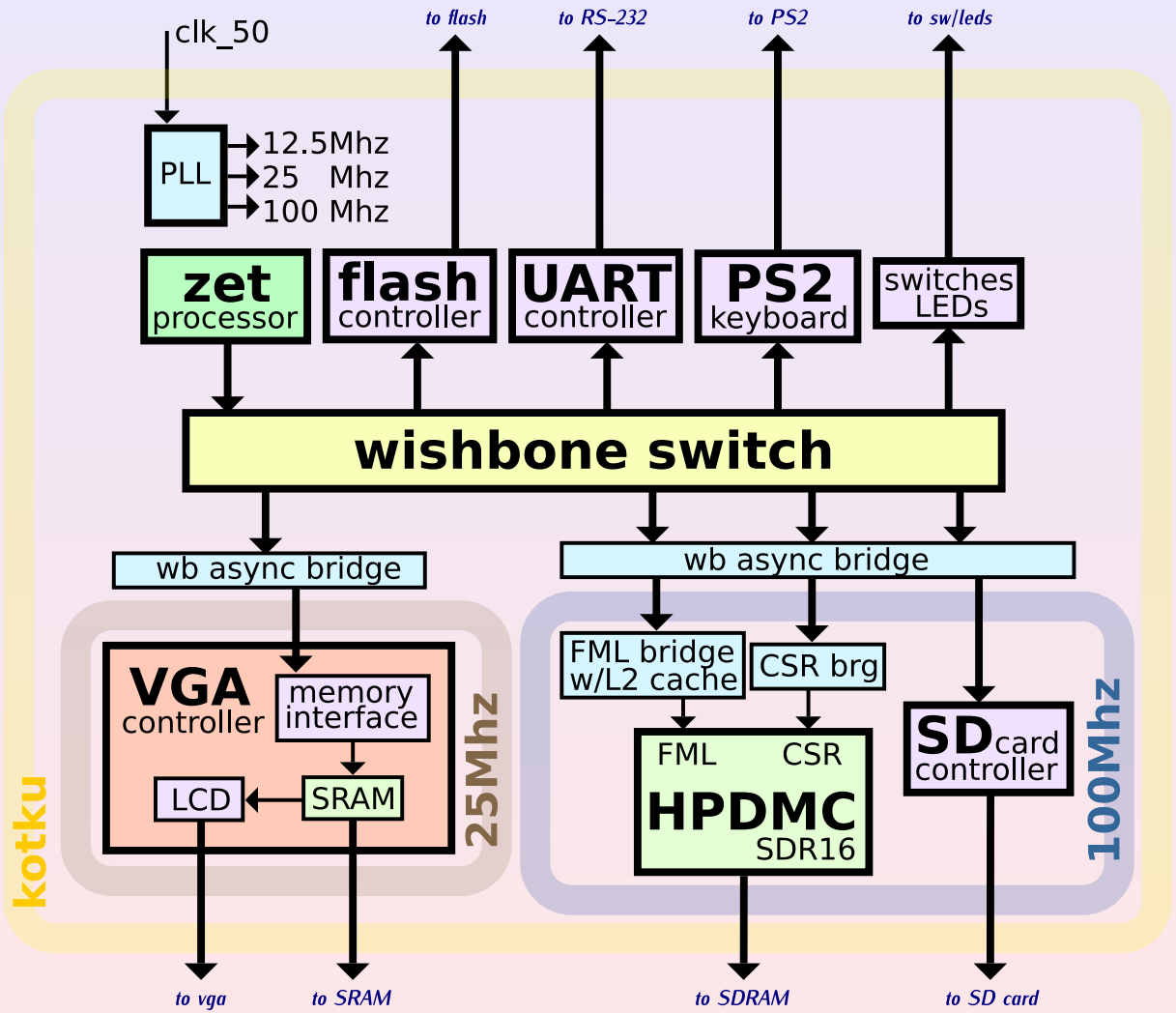
FPGA physical layout

- LE: 9,397 / 18,752 (50 %)
- PINs: 202 / 315 (64 %)

Contents

Zet SoC FPGA toplevel

version 1.1



Wishbone master

Zet x86 proc

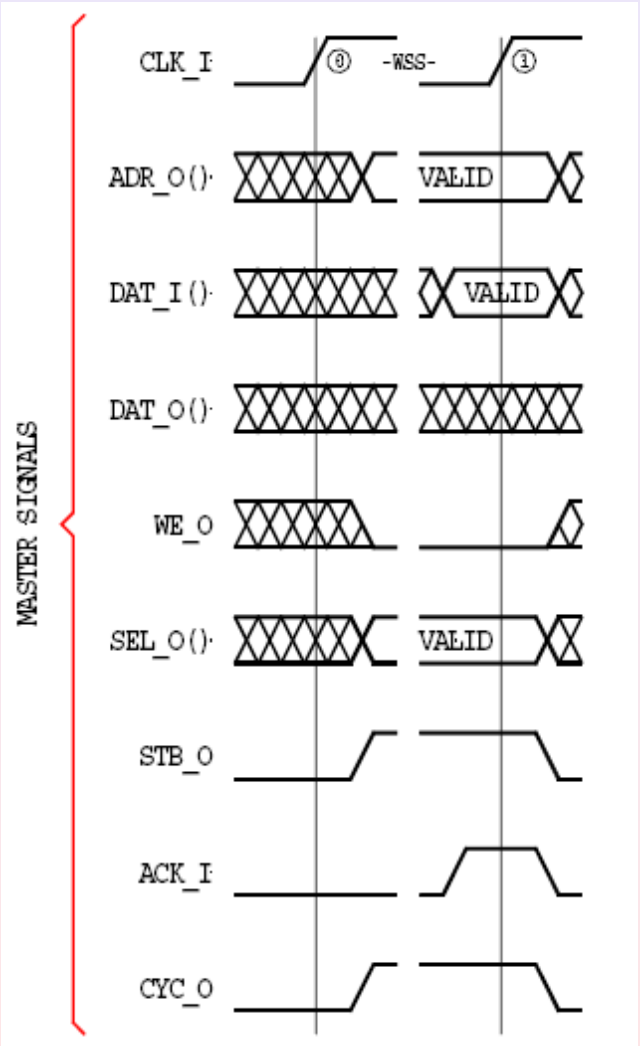
Wishbone slaves

1. Flash
2. UART
3. PS2 keyb
4. sw LEDs
5. VGA
6. FML bridge to RAM
7. SD card

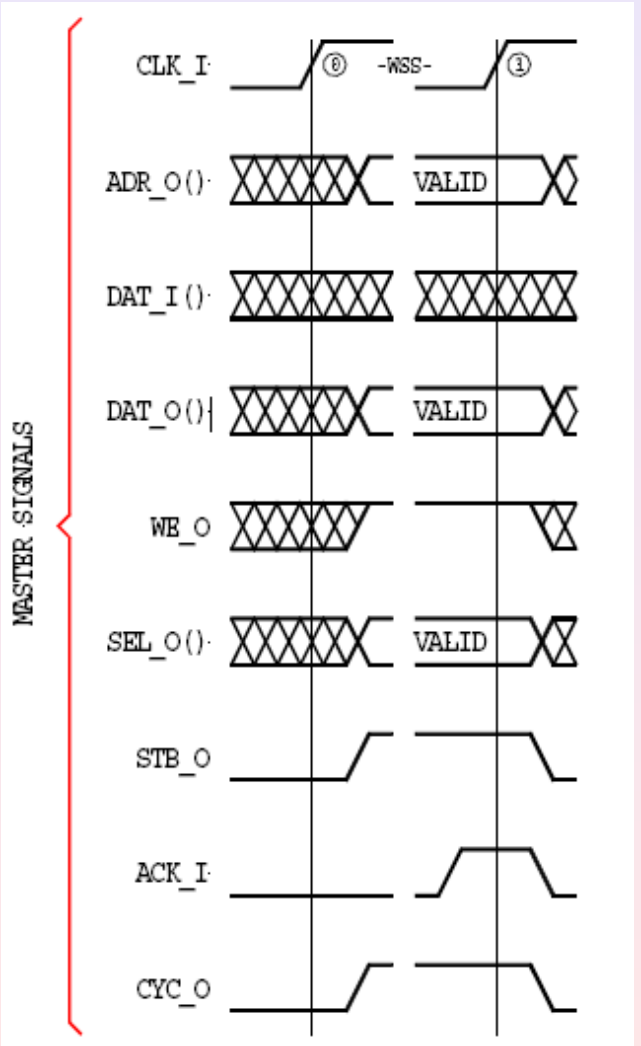
Wishbone SoC interconnection architecture

Single read/write transactions

Read:

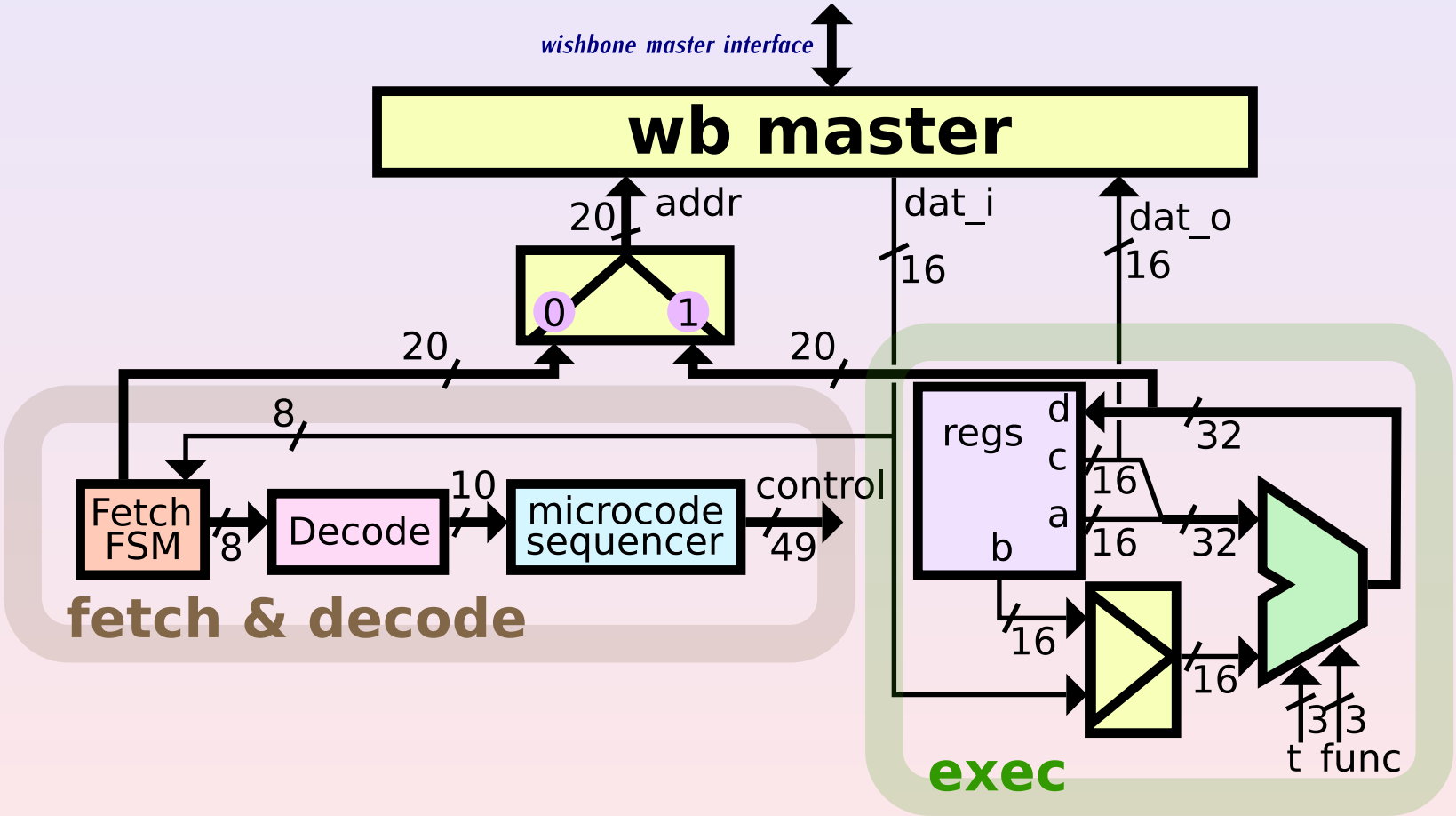


Write:

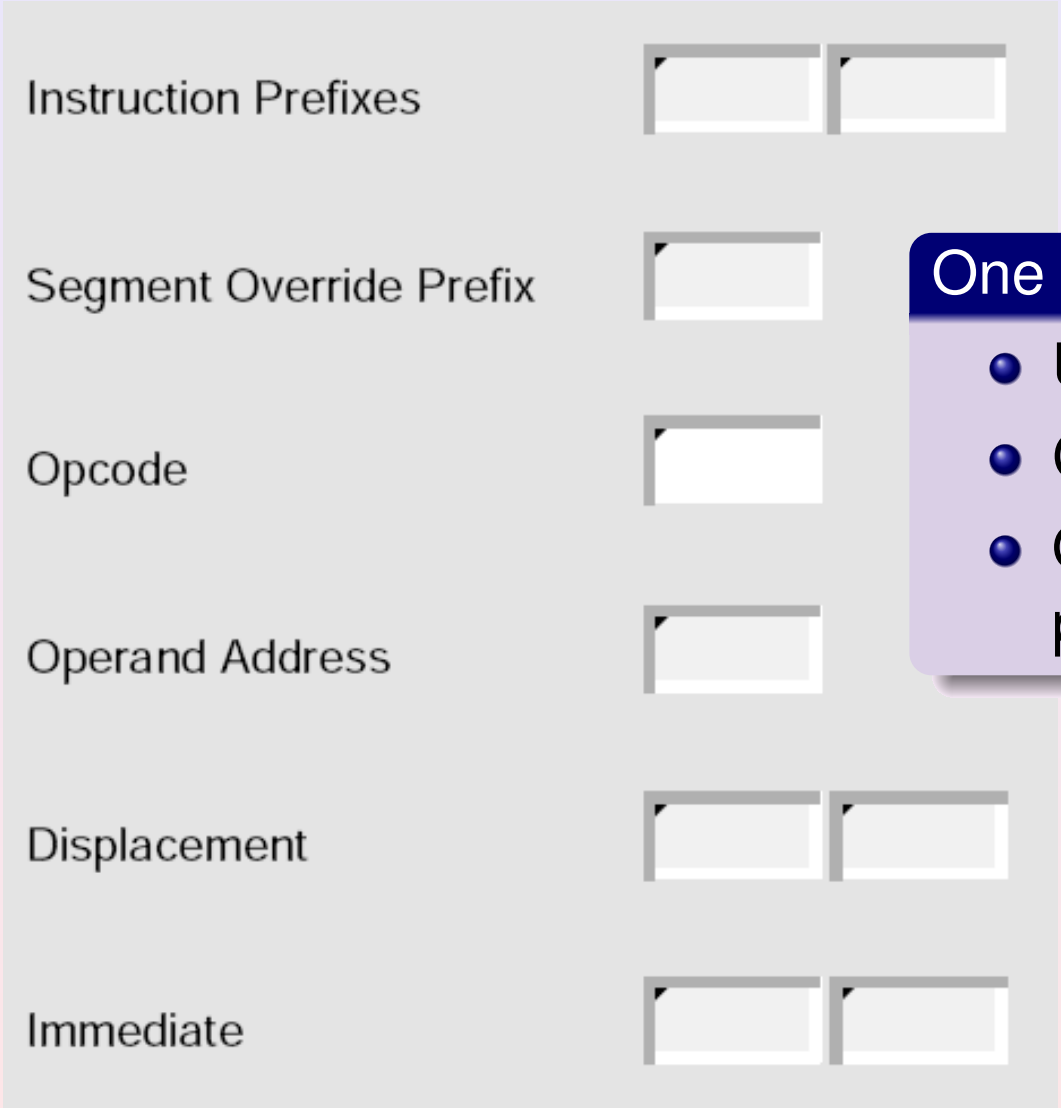


Zet processor toplevel

Simplified block schematic for version 1.1



8086 instruction format



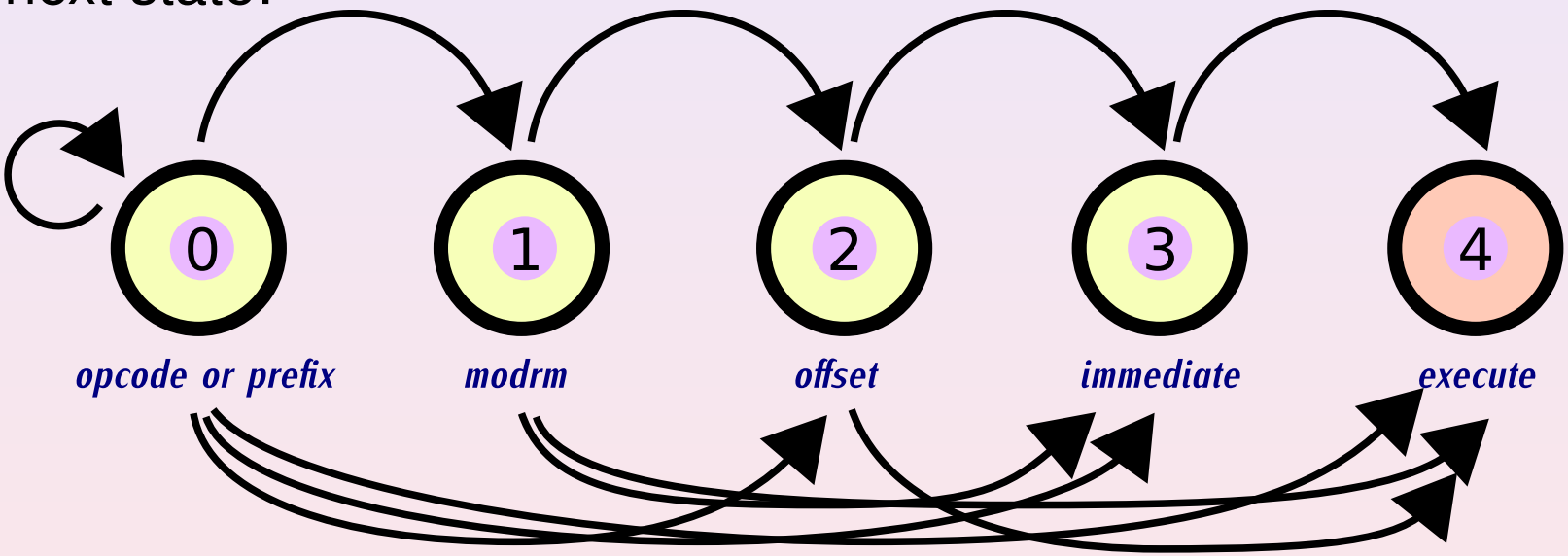
One single instruction

- Up to 9 bytes in length!
- Can be 1 byte long
- Opcode is always present

Fetch FSM

Decoder tells what to do

Each cycle we progress through the FSM, decoder decides next state:



16-bit x86 instruction examples

CISC in its full extent

encoding of “lock movw $0x7432,%cs:-0x101(%bx,%di)”

1. **0x2e** (prefix): cs
2. **0xf0** (prefix): lock
3. **0xc7** (opcode): movw
4. **0x81** (modrm): (%bx,%di)
5. **0xff 0xfe** (offset): -0x101
6. **0x32 0x74** (immediate): 0x7432

encoding of “push %bx”

1. **0x53** (opcode + operand): push %bx

16-bit x86 instruction examples (2)

CISC in its full extent

instruction “**into**”, probably one of the most complex:

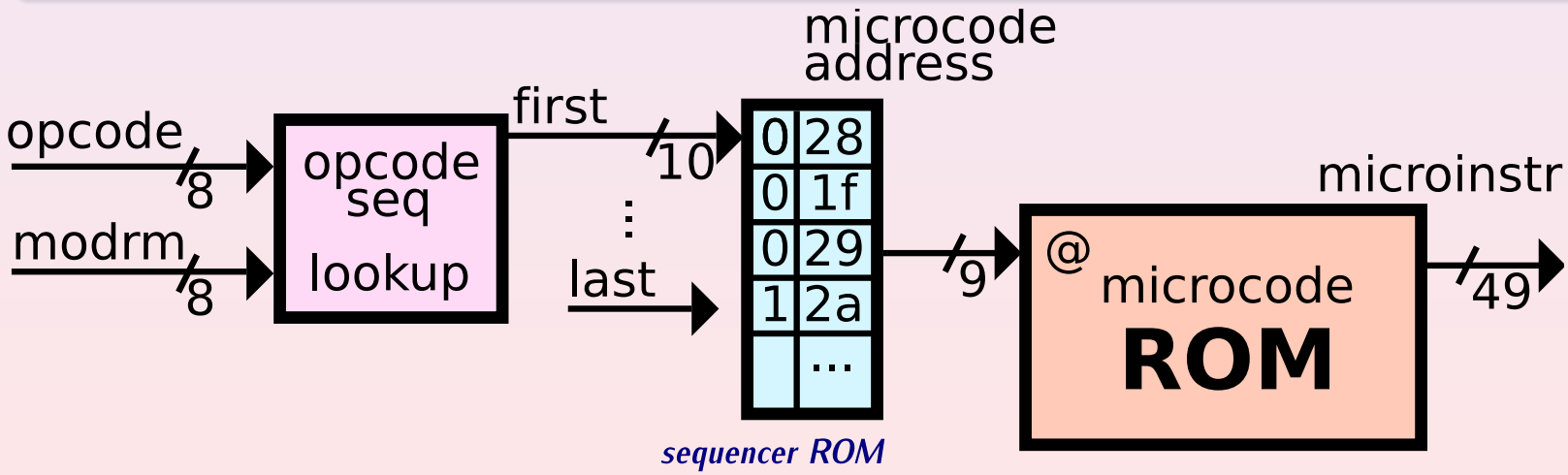
1. Check if OF == 1 then
2. SP = SP - 2
3. flags to stack
4. IF = 0; TF = 0
5. SP = SP - 2
6. CS to stack
7. CS = *((unsigned short) 0x12)
8. SP = SP - 2
9. IP to stack
10. IP = *((unsigned short) 0x10)

Decoder

Opcode lookup - sequencer ROM - microcode ROM

Decode stage

1. From “opcode” and “modrm” we get a sequencer address. This is implemented in logic to minimize memory utilization.
2. The sequencer keeps reading in the ROM till it reaches “1”
3. For each address in the sequencer, use it to index the microcode ROM to get the actual microinstruction.



Microcode format

bit	name	description
1:0	**addr_s**	Register address (read) for the segment that goes to *regfile*
5:2	**addr_a**	Register address A (read) that goes to *regfile*
9:6	**addr_b**	Register address B (read) that goes to *regfile*
13:10	**addr_c**	Register address C (read) that goes to *regfile*
17:14	**addr_d**	Register address D (write) that goes to *regfile*
18	**wrfl**	Write to flags?
19	**wr_mem**	Write to mem?
20	**wr**	Write to *regfile*?
21	**wr_cnd**	Conditional write to *regfile*
22	**high**	Write to *regfile* for bits 31:16 that comes out from the ALU to DX (mul/div operations).
25:23	**t**	Instruction type (controls the main ALU's multiplexer)
28:26	**func**	Function inside the type (second multiplexer, inside each module type in the ALU)
29	**byteop**	Byte operation
31:30	**memalu**	What goes to *regfile* in the D bus is the memory output or the ALU?
32	**m_io**	Is this an IO address or MEM address?
33	**b_imm**	Control to multiplexer to choose if that goes to the ALU from bus B comes from *regfile* or from the immediate
34	**a_byte**	The *addr_a* address refers to a 8 bit register?
35	**c_byte**	The *addr_c* address refers to a 8 bit register?
36	**var_s**	Control from the variable field *addr_s*, described below.
38:37	**var_a**	Control from the variable field *addr_a*, described below.
40:39	**var_b**	Control from the variable field *addr_b*, described below.
42:41	**var_c**	Control from the variable field *addr_c*, described below.
44:43	**var_d**	Control from the variable field *addr_d*, described below.
45	**var_off**	Control from the variable field *offset*, described below.
48:46	**var_imm**	Control from the variable field *imm*, described below.

Microcode format

Previous example: “into”

```
######  vi vo vd vc vb va vs cb ab im mi ma by fun  t  wh wr wm wf ad_d ad_c ad_b ad_a  s
INTO   xxx__x_xx_xx_xx_xx__x__x__x__x__x_0x__x_111_010__0_00__0__0_xxxx_xxxx_xxxx_xxxx_xx  // o?
INTO   001__x_00_xx_xx_00__x__x__0__1__x_01__0_101_001__0_01__0__0_0100_xxxx_xxxx_0100_xx  // sp-2
INTO   000__x_00_xx_xx_xx__x__x__x__1__x_01__0_101_111__0_01__0__0_1101_xxxx_xxxx_xxxx_xx  // f->r
INTO   xxx__0_xx_00_00_00__0__0__0__0__0_1x__0_000_111__0_00__1__0_xxxx_1101_1100_0100_10  // st(fl)
INTO   xxx__x_xx_xx_xx_xx__x__x__x__1__x_01__0_110_111__0_00__0__1_xxxx_xxxx_xxxx_xxxx_xx  // i=0 t=0
INTO   010__x_00_xx_xx_00__x__x__0__1__x_01__0_101_001__0_01__0__0_0100_xxxx_xxxx_0100_xx  // sp-4
INTO   xxx__0_xx_00_00_00__0__0__0__0__0_1x__0_000_111__0_00__1__0_xxxx_1111_1100_0100_10  // st(ip)
INTO   xxx__0_xx_00_00_00__0__0__0__0__0_1x__0_001_111__0_00__1__0_xxxx_1001_1100_0100_10  // st(cs)
INTO   010__x_00_xx_xx_xx__x__x__0__1__x_01__0_000_000__0_01__0__0_1101_xxxx_xxxx_xxxx_xx  // 4->tmp
INTO   001__x_00_xx_xx_00__x__x__0__1__x_01__0_000_101__0_01__0__0_1101_xxxx_xxxx_1101_xx  // tmp*4
INTO   001__x_00_xx_xx_00__x__x__0__1__0_10__0_001_001__0_01__0__0_1001_xxxx_xxxx_1101_xx  // ld(cs)
INTO   000__x_00_xx_xx_00__x__x__0__1__0_10__0_001_001__0_01__0__0_1111_xxxx_xxxx_1101_xx  // ld(ip)
```

There is a script **bin/web2rom** that fetches all the microcode definitions and creates the corresponding ROMs files.

Simulation

(simulation example)

Zet New Generation

New features

- Get rid of the Fetch FSM. Consider pref as normal insn
- 100 Mhz instead of 12.5 Mhz. Then remove asynchronous bridges
- Max 1 IPC (now it's about 4-6 IPC)
- iCache and write-through dCache. Use 2 wb masters at the toplevel.

How to do it

- Design each stage separatelly, registering outputs
- Create a stub for each stage to check max 9.5ns routing delay
- Use already written test suites for each instruction type

Proposed pipeline

8 stage pipeline

Description of the pipeline

8 stage pipeline:

- **Fetch**. Fetch 6 bytes from memory
- **Decode**. Calculate the sequencer address, based on the opcode, mode byte. Calculate instruction size, feed it back to fetch.
- **Sequencer**. Calculate the microcode address, get it from the sequencer ROM.
- **Issue**. Get the microcode IR from the microcode ROM.
- **Read**. Read from the register file
- **Execute**. ALU execution: sum, mul, div, eff addr, ...
- **Memory**. Load/store from memory.
- **Write back**. Write to register file.