μ[Dark]RISC

The secret steps before DarkRISCV!

Marcelo Samsoniuk — Darklife Research

μ[Dark]RISC

Features

Designed back in 2015 (years before DarkRISCV), it includes:

- Two state pipelined RISC
- Flexible 16x16-bit registers
- 16-bit Program Counter (PC)
- Harvard Architecture w/ separate ROM/RAM up to 64Kword each
- Memory mapped IO
- Less than 100 lines of Verilog
- 1/3 of a XC3S50AN@75MHz
- Can be easily changed!

μ[Dark]RISC
History & Motivation

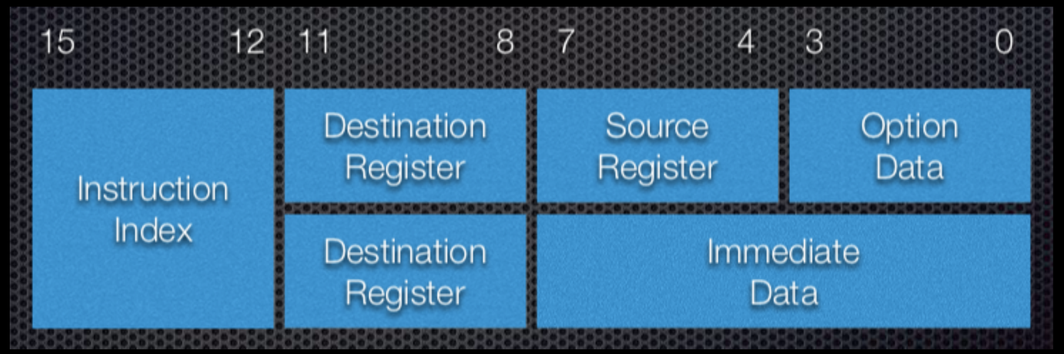
- The beggning: VLIW DSPs on FPGAs: high optimized ALUs around DSP blocks, in a way that MAC operations can be optimized. However, conventional code was hard to port for VLIW DSPs, so a more general purpose processor was needed…

- Lots of different concepts around accumulator-oriented, register bank oriented, VLIW, 16/32-bits, etc most designs were identified just as “core” or “dsp”, but this specific concept was named uRISC for an external presentation on a University. Because there are too much processors called uRISC already, it was renamed to uDarkRISC (micro-DarkRISC).

- Designed as a evaluation processor, it was never designed to be used on real products and never tested with complex applications… however, the more conventional approach was used as base for DarkRISCV, a high-performance processor which implements a RV32I/E 100% compatible with GCC compiler!

μ[Dark]RISC
Instruction Decode

The 16bit instruction word is read from the synchronous BRAM and divided in 3 or 4 fields:




```
wire          [ 3:0]  INST =    IDATA[15:12];
wire          [ 3:0]  DPTR =    IDATA[11: 8];
wire          [ 3:0]  SPTR =    IDATA[ 7: 4];
wire          [ 3:0]  OPTS =    IDATA[ 3: 0];

wire signed [15:0]  IMMS = { IDATA[7]?8'hff:8'h00, IDATA[7:0] };
wire signed [15:0]  DREG = REG[DPTR];
wire signed [15:0]  SREG = REG[SPTR];
wire signed [15:0]  DMUX   [0:15];
```

The focus here was fit two cores on a cheap Spartan-3 50AN FPGA!

μ[Dark]RISC
Instruction Execution

According to the Destination Register, Source Register, Option Data or Immediate Data, all 16 possible instructions are computed in parallel:

```
assign DMUX[`ROR] = SREG>>>OPTS;                  // dreg = sreg>>>opts
assign DMUX[`ROL] = SREG<<<OPTS;                  // dreg = sreg<<<opts
assign DMUX[`ADD] = (OPTS?DREG+OPTS:DREG+SREG);   // dreg = dreg+opts or dreg+sreg
assign DMUX[`SUB] = (OPTS?DREG-OPTS:DREG-DREG);   // dreg = dreg-opts or dreg-sreg
assign DMUX[`XOR] = DREG^SREG;                    // dreg = dreg xor sreg
assign DMUX[`AND] = DREG&SREG;                    // dreg = dreg and sreg
assign DMUX[ `OR] = DREG|SREG;                    // dreg = dreg or  sreg
assign DMUX[`NOT] = ~DREG;                        // dreg = not dreg
assign DMUX[`LOD] = DATA;                         // dreg = *sreg
assign DMUX[`STO] = DREG;
assign DMUX[`IMM] = IMMS;                         // dreg = imms
assign DMUX[`MUL] = (DREG*SREG)>>>OPTS;           // dreg = (dreg*sreg)>>>opts
assign DMUX[`BRA] = DREG;
assign DMUX[`BSR] = PC+1;                         // dreg = pc+1, pc += imms
assign DMUX[`RET] = DREG;
assign DMUX[`LOP] = DREG-1;                       // dreg = dreg-1
```

μ[Dark]RISC
Instruction Execution

According to the Destination Register, Source Registe mmediate Data, all 16 possible instructions are computed in parallel:

```
assign DMUX[`ROR] = SREG>>>OPTS;
assign DMUX[`ROL] = SREG<<<OPTS;
assign DMUX[`ADD] = (OPTS?DREG+OPTS
assign DMUX[`SUB] = (OPTS?DREG-OPTS
assign DMUX[`XOR] = DREG^SREG;
assign DMUX[`AND] = DREG&SREG;
assign DMUX[ `OR] = DREG|SREG;
assign DMUX[`NOT] = ~DREG;
assign DMUX[`LOD] = DATA;
assign DMUX[`STO] = DREG;
assign DMUX[`IMM] = IMMS;                    // dreg = imms
assign DMUX[`MUL] = (DREG*SREG)>>>OPTS;      // dreg = (dreg*sreg)>>>opts
assign DMUX[`BRA] = DREG;
assign DMUX[`BSR] = PC+1;                    // dreg = pc+1, pc += imms
assign DMUX[`RET] = DREG;
assign DMUX[`LOP] = DREG-1;                  // dreg = dreg-1
```

reg+sreg
dreg-sreg
g
eg
sreg
reg
*sreg

Although supposed to be “general purpose”, it includes a MAC instruction for multiply with shift. Some variations included a separate 32-bit accumulator, but it was removed…

μ[Dark]RISC
Instruction Execution

According to the Destination Register, Source Register, Option Data or Immediate Data, all 16 possible instructions are computed in parallel:

```
assign DMUX[`ROR] = SREG>>>OPTS;
assign DMUX[`ROL] = SREG<<<OPTS;
assign DMUX[`ADD] = (OPTS?DREG+O
assign DMUX[`SUB] = (OPTS?DREG
assign DMUX[`XOR] = DREG^SREG;
assign DMUX[`AND] = DREG&SREG;
assign DMUX[ `OR] = DREG|SREG;
assign DMUX[`NOT] = ~DREG;
assign DMUX[`LOD] = DATA;
assign DMUX[`STO] = DREG;
assign DMUX[`IMM] = IMMS;
assign DMUX[`MUL] = (DREG*SREG)>>>
assign DMUX[`BRA] = DREG;
assign DMUX[`BSR] = PC+1;
assign DMUX[`RET] = DREG;
assign DMUX[`LOP] = DREG-1;
```

```
eg>>>opts
<<opts
ts or dreg+sreg
s or dreg-sreg
sreg
sreg
sreg
s
(dreg*sreg)>>>opts
// dreg = pc+1, pc += imms
// dreg = dreg-1
```

No conditional branches other than a single “test and decrement” instruction… basically, a kind of "jumps when the register is not negative”, a very bad decision!

μ[Dark]RISC
Write Registers

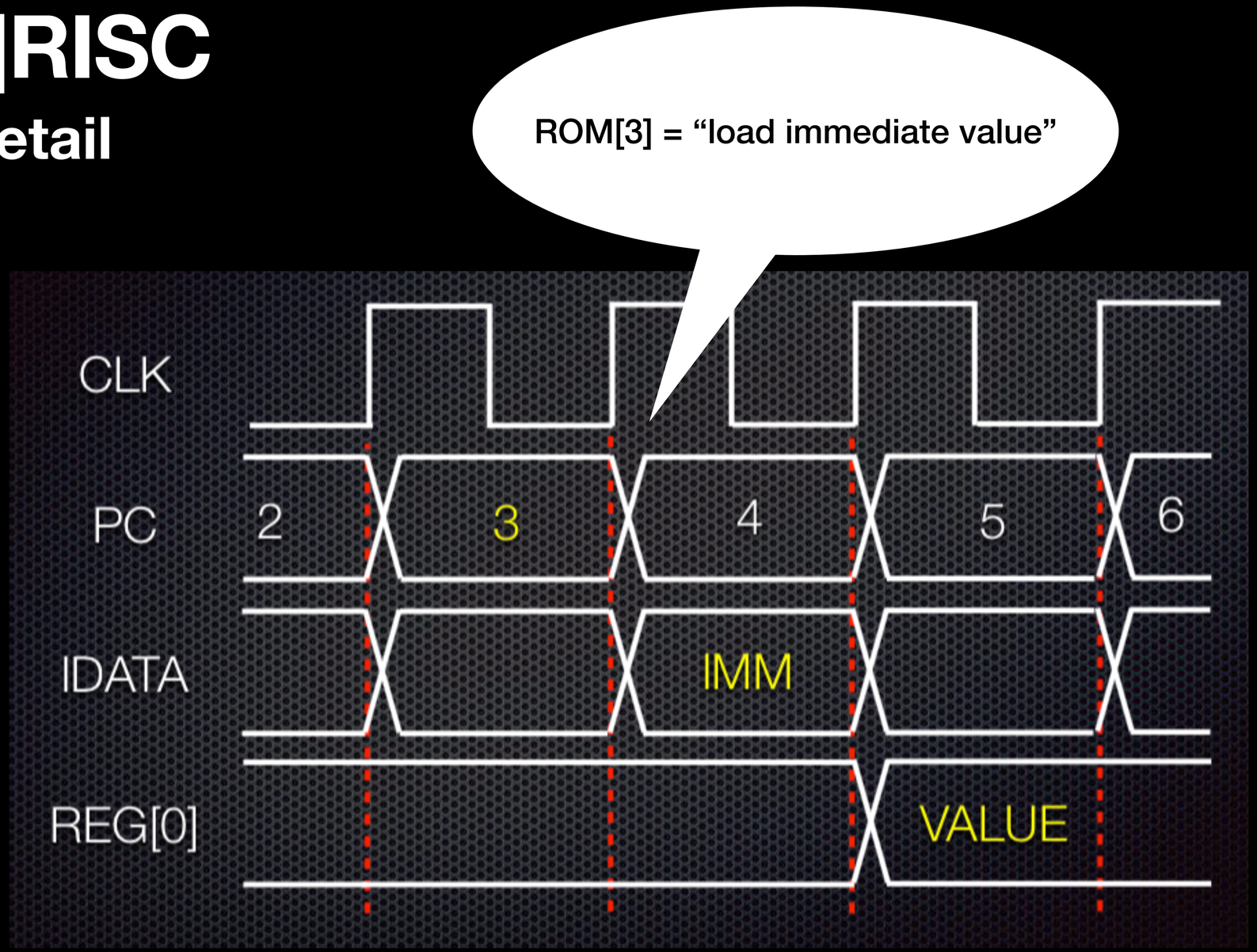
According to the Instruction Index, the one computed result is selected from the instruction array and the result stored in the Destination Register:

```
always@(posedge CLK)
begin
    REG[DPTR] <=    DMUX[INST];

    PC <=           !RES ? 0 :
             INST==`RET ? DREG :
                          PC+(INST==`BSR||INST==`BRA||(INST==`LOP&&!DREG[15])?IMMS:1);
end

assign RD     = INST==`LOD;
assign WR     = INST==`STO;
assign DATA   = RD ? 16'hzzzz : DREG;
assign ADDR   = SREG;
```

μ[Dark]RISC
Pipeline Detail



μ[Dark]RISC
Basic Instruction Set

The direct supported instructions are:

- ROR/ROL: register right or left rotate
- ADD/SUB: register add/sub
- XOR/AND/OR/NOT: register logic operations
- LOD/STO: register to/from memory operations
- IMM: immediate data load (8-bit LSB value)
- MUL: register multiply and right rotate (trying include some DSP support, but without a wide accumulator)
- BRA/BSR: load PC from PC+immediate data and, optionally, save PC to a register
- RET: load PC from register data
- LOP: test register, decrement register and set PC to PC+immediate data

μ[Dark]RISC

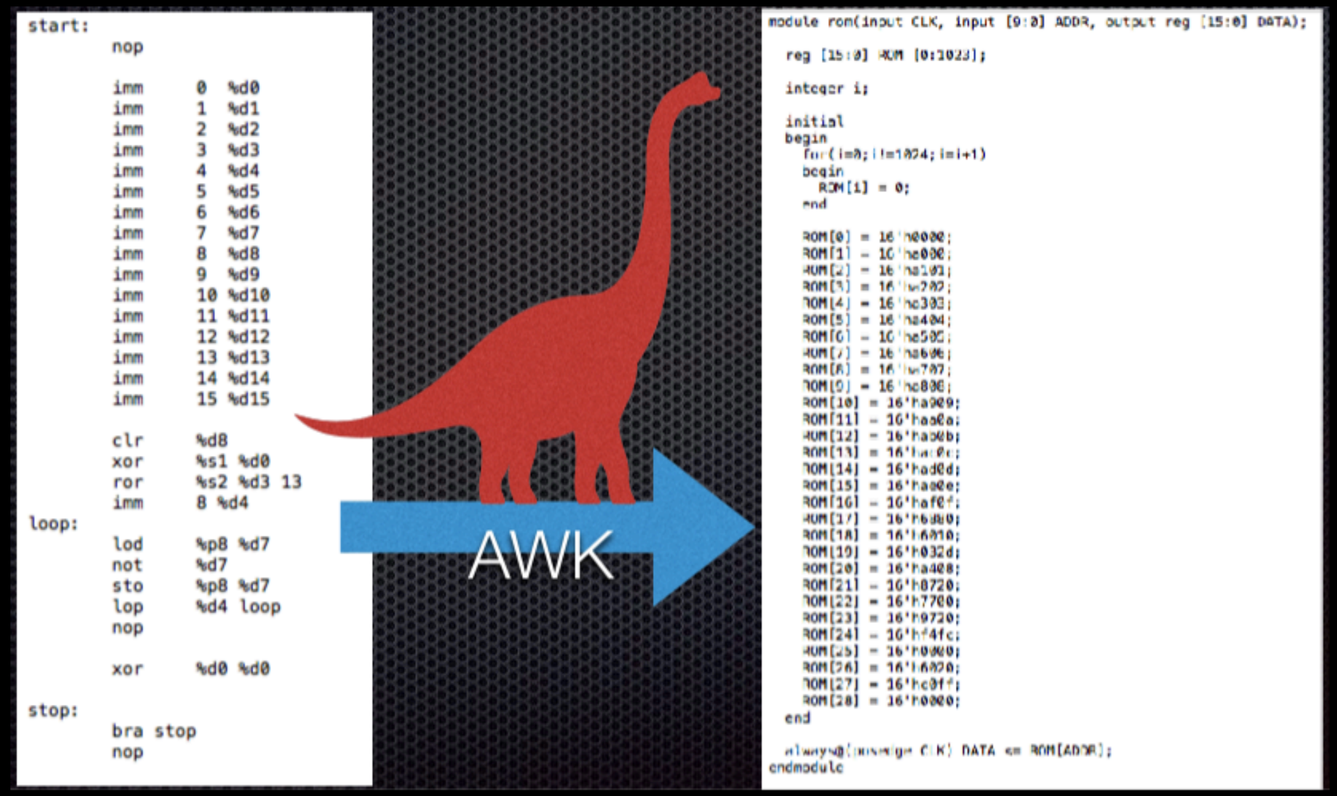
Advanced Instruction Set (aka “pseudo-instructions”)

The known pseudo-instructions are:

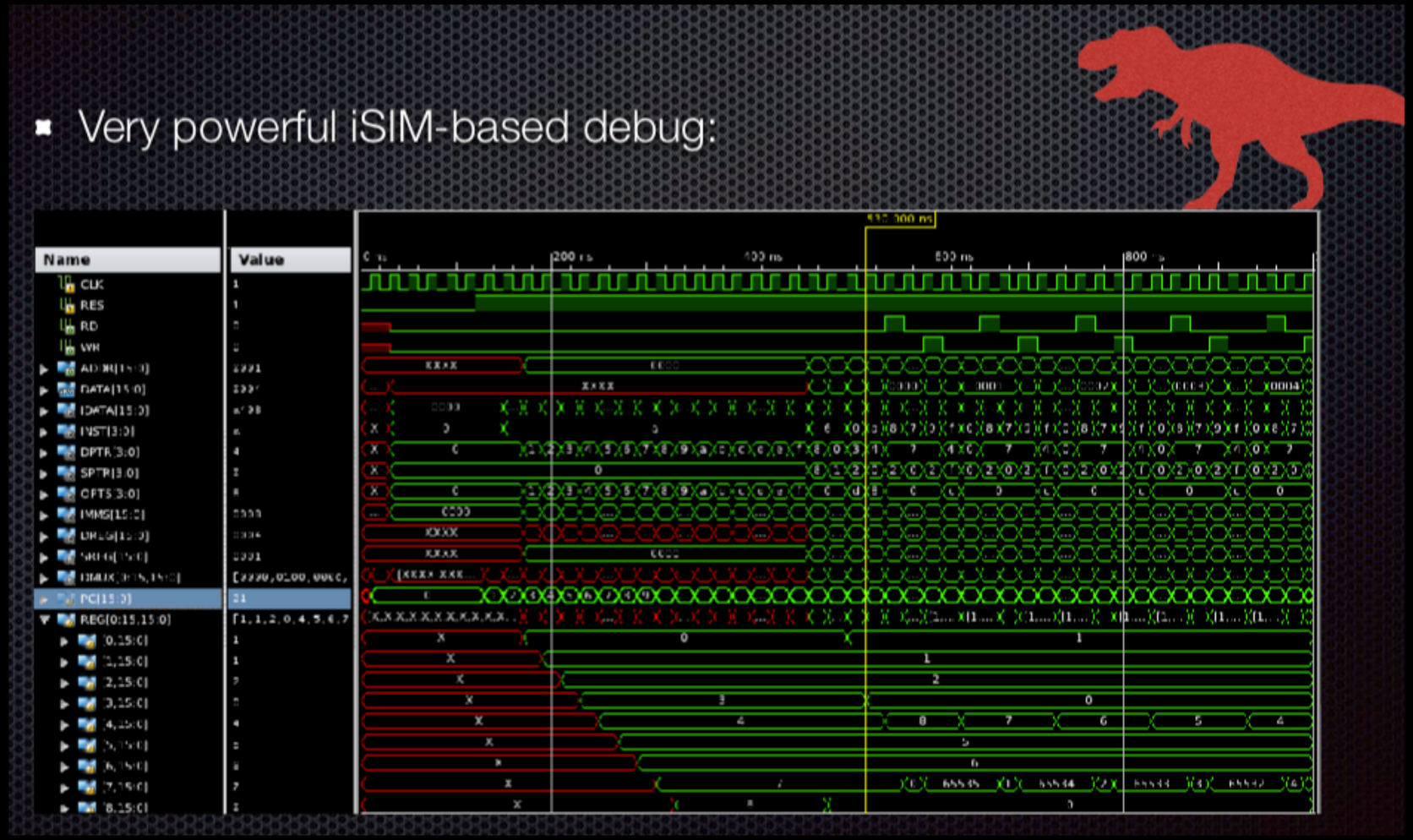
- MOV: ROR or ROL with shift value = 0.
- NOP: MOV with same source/destination register.
- ADDQ/SUBQ: quick add/sub a value between 1 and 15.
- JMP: move the address to a register and RET from this register
- INC/DEC: ADDQ/SUBQ with value 1.
- CLR: xor in the same register.
- and much more!

μ[Dark]RISC
Development System

Limited to an AWK-based assembler that generates a Verilog file w/ the ROM description:



μ[Dark]RISC
Development System



μ[Dark]RISC

Conclusion

Although the uDarkRISC was frozen in time and replaced by the DarkRISCV, there are always people trying find a simple processor to study, which means that there is no bad processors, just processors that are better for different applications. So, although DarkRISCV is good because it can actually run complex code generated by GCC, the uDarkRISC may be far better for starters that are just trying understand how design a simple processor.