Design-time customisation and generation of reconfigurable processors

Jeroen Leijten
co-founder and chief processor architect
jeroen.leijten@philips.com

Silicon Hive, Philips Technology Incubator, WDB11, Prof. Holstlaan 4, 5656AA Eindhoven, The Netherlands
info@siliconhive.com, http://www.siliconhive.com

Outline

• Introduction
• Processor architecture template
• Processor design methodology
• Processor specification
• Processor generation
• Examples
• Summary
• Demo
Introduction

Silicon Hive’s Reconfigurable Accelerators aim: to enable computationally efficient high-level programmable systems on chip (SoC) through coarse-grained reconfigurable accelerators.

Traditional processor architecture

Function units (e.g. ALUs, multipliers) perform the actual computation.

Register files and data memories coupled to load/store units store data and temporary values.

Silicon Hive

Computational efficiency (MOPS/Watt)

Replacing ASIC blocks

Hardwired accelerators (ASICs)

Traditional DSPs (e.g. TI ‘C54, R.E.A.L.)

Low-end embedded GPPs (e.g. ARM, MIPS)

High-performance GPPs (e.g. Intel x86)

Feature size (nm)


350 130 65 45

1997 1999 2004

32 22

2013 2016 year

Silicon Hive

Silicon Hive

Silicon Hive
Unlike ASICs, traditional processors have considerable computationally inactive overhead due to control and synchronisation.

Flexibility limited by hardware-fixed instruction set.
Traditional processor architecture

Centralised resources and full connectivity prevent scalability

Dramatically reduce control overhead, expose all pipeline management to the instruction set, move complexity to the compiler.

Compiler must explicitly schedule all pipeline stages!
ULIW architecture template

Remove full connectivity. Use point-to-point, partially connected networks. Things become more scalable…
ULIW architecture template

Partition and distribute all resources

? Massively parallel architecture

Use many local data memories to improve locality of reference, increase bandwidth, and reduce bus traffic

? Higher performance and lower power

Exposé all control points in the datapath to the instructions. Add flexibility to do all combinations the datapath can support (and there are many!)

? Ultra long instructions

ULIW
ULIW architecture template

High locality of reference & regularity +
Massive parallelism +
Very low control overhead =
Very high computational efficiency (MOPS/W)!!!

Kernel program schedule

Standard DSP mode. One instruction/cycle. Typically only a part of the datapath is utilised. Used for pre- and post-ambles, conditional and irregular code, etc.
Kernel program schedule

Pure data-flow mode. Entire inner-loop body is scheduled onto one instruction (configuration). It’s like an FPGA! Data ripples through on clock tick, but configuration remains!

Architecture components

Cell (complete processor)

Processing and Storage Element, or PSE (data-path segment)
**Architecture dimensions**

*Processing and Storage Element (PSE)*

- **CL**: }
- **RF**: }
- **IN**: }
- **FU**: }
- **MEM**: }
- **LD/ST**: }
- **IS**: }

**Cell**

**Streaming array**

---

**Different instances**

- **MOUSTIQUE block accelerator** (ONE CELL, ONE PSE)
  - Framed data in memory for highly cost-sensitive apps. “Smaller than ASIC”

- **AVISPA block accelerator** (ONE CELL, MULTIPLE PSEs)
  - Framed data in memory (e.g. OFDM demodulation)

- **BRESCA stream accelerator** (MULTIPLE CELLS, ONE PSE EACH)
  - Streaming data, very high/dynamic rates (e.g. IF front-end filtering)

---

**Instances are not handwritten, but are generated from internal design methodology**
Processor design methodology

- TIM machine description
- High-level C program
- State view & trace file
- HDL code
- Logic synth., place & route
- Netlist layout
- Function unit library
- Processor simulator/generator
- Binary code
- Assembler & linker
- Assembly code (C-syntax)
- Standard C compiler
- Compiled simulator
- Operation semantic library
- Processor model generator
- Processor model (C-syntax)
- Netlist layout
- Area, speed, power
- Cycle count

The design methodology enables finding the proper match of processor architecture, compiler, and target application quickly.
Genesys Basic Concepts

based on highly optimized ultra-configurable RT-level hardware building blocks

- core IO architecture template
- core architecture template
- function units (ALU, MAC, LSU, ...)
- basic blocks (logic gates, clock gates, flipflops, latches, ...)

- cell architecture template
- other generic parts (register files, memories, ...)

- arithmetic blocks (adders, multipliers, shifters, ...)

proprietary C++-based hardware generation and simulation kernel

Genesys HDL Generation

- TIM machine description
  - highly abstract
  - one file configures all Silicon Hive tools (Genesys, HiveCC, SHAPE, simulators, debugger)
  - typically only $O(100)$ lines for complete cell

- program switches

- automatically generated synthesizable HDL
  - no manual editing required
  - well-formatted and well-structured
  - correct by construction
  - typically $O(10000)$ lines for complete cell

productivity increase $O(100)$!!!

drastic changes in just minutes!!!
TIM description fragment

/* port types */
Port Port32 { Width = 32; };
...

/* register file types */
RF RF32r8 (Port32 ip0) -> (Port32 op0) {
    Width = 32; Latency = 1; Capacity = 8;
};
...

/* operation semantics */
OP passh (wsigned a) -> (wsigned b) {
    SEM b = a << (width_b-width_a);
};
...

/* function unit types */
FU psu_passonly_imm (Port32 ip) -> (Port32 op) {
    imm: op = pass(Immediate(ip,[-32768, .., 32767]));
    immh: op = passh(Immediate(ip,[0, .., 65535]));
    pass: op = pass(ip);
};
...

/* issue slot types */
IS slot2 (Port32 ip0, ip1, ip2) -> (Port32 op) {
    lsu lsu(ip0, ip1, ip2); op = lsu.op;
    psu_psuonly_imm_1 psu(ip0); op = psu.op;
};
...

/* cell description */
#define ALL s1.op, s2.op, s3.op, s4.op, s1.dlc
Machine moustique {
    program_counter PC (s1.ja);
    status_register SR (s1.usw);
    RF32r8 rf1 ({ALL});
    RF32r8 rf2 ({ALL});
    RF32r8 rf3 ({ALL});
    RF32r8 rf4 ({ALL});
    RF32r8 rf5 ({ALL});
    RF32r8 rf6 ({ALL});
    RF32r8 rf7 ({ALL});
    RF64r2 rf64  ({s3.op64, s4.op64});
    RF64r4 rf64b ({s3.op64, s4.op64});
    slot1 s1 (rf1.op0, rf2.op0, rf5.op0, PC.rp, SR.rp);
    slot2 s2 (rf3.op0, rf4.op0, rf5.op0);
    slot3 s3 (rf3.op0, rf4.op0, rf5.op0, rf64.op0);
    slot4 s4 (rf6.op0, rf7.op0, rf64b.op0);
}
Creating RtSys models

- CHDL: proprietary Configurable Hardware Description Language
- VHDL-inspired syntax and semantics
- serves as frontend language to RtSys
- supports highly configurable constructs
  - control over generation of ports, signals, variables, processes, child modules, etc.
  - all statements can be guarded
  - built-in multidimensional array support
  - external C++-function support
  - supports basic inheritance

CHDL fragment

ENTITY RtfFunctionUnit IS
PARAM |
  p_slot_id : FEATURE INTEGER;
  p_fu_id : FEATURE INTEGER;
  p_use_optype : FEATURE D1-INTEGER;
p_input_data_bits : GENERIC D1-INTEGER;
p_max_output_delay : CONSTANT INTEGER;
p_optype_bits : GENERIC INTEGER;
  ...
PORT ( WITH p_max_output_delay > 0 GUARD
  in_stage_clock : IN D1-BIT(p_max_output_delay);
  in_reset : IN BIT;
in_optype : IN BITVECTOR(p_optype_bits);
in_input_data : IN D1-BITVECTOR( p_input_data_bits;
  out_output_data : OUT D1-BITVECTOR( p_output_data_bits);
  ...
); END;
EXTERN RtfPSUOptypeSet;
ARCHITECTURE RtfPSU OF RtfFunctionUnit IS
  ...
BEGIN ...

Built-in array types
Configurable ports
Guarding of statements
Inheritance
External C++ function calls
void RtfPSU::decode_execute() {
    if (p_optype_bits->value() > 0) {
        *var_optype = RtsToUnsignedInt(RtsCastUnsignedVector(
            in_optype->read()));
    } else {
        *var_optype = 0;
    }
    ...
}

void RtfPSU::decode_hdl() {
    if (p_optype_bits->value() > 0) {
        fa() <<
            RtsHdlVarAssign(var_optype->name(),
            RtsHdlToUnsignedInt(RtsHdlCastUnsigned(
                in_optype->name())));
    } else {
        fa() <<
            RtsHdlVarAssign(var_optype->name(), "0");
    }
    ...
}

RtfPSU::RtfPSU(
        const string &name,
        const RtsParam &p_slot_id,
        const RtsParam &p_fu_id,
        const Rts1DArray<RtsParam> &p_use_optype,
        const Rts1DArray<RtsParam> &p_input_data_bits,
        const Rts1DArray<RtsParam> &p_output_data_bits,
        RtsModule &parent_module
    ) : RtsFunctionUnit(name,
        p_slot_id,
        p_fu_id,
        ...
    parent_module) {
    var_optype = new RtsVarUInt(
        "var_optype", p_optype_bits, this);
    var_output_data = new Rts1DArray<RtsVarBoolVector>(
        "var_output_data", p_output_data_bits, this);
    ...
    begin();
    RTS_BEGIN_PROCESS("decode", decode);
    decode->sensitive(*in_optype);
    decode->sensitive(*in_input_data);
    decode->variable(*var_optype);
    decode->variable(*var_output_data);
    RTS_END_PROCESS(decode);
    ...
    end();
}

Inheritance
HDL generation of behaviour
Simulation of behaviour

Generated VHDL fragment

ENTITY hive_fu_rtfpsu IS
    GENERIC ( --
        p_input_data_bits_I0 : INTEGER := 16;
        p_input_data_bits_I1 : INTEGER := 16;
        p_output_data_bits_I0 : INTEGER := 16;
        p_optype_bits : INTEGER := 2;
    );
    PORT ( --
        in_optype : IN std_logic_vector(p_optype_bits – 1 DOWNTO 0);
        in_input_data_I0 : IN std_logic_vector( p_input_data_bits_I0 – 1 DOWNTO 0);
        in_input_data_I1 : IN std_logic_vector( p_input_data_bits_I1 – 1 DOWNTO 0);
        out_output_data_I0 : OUT std_logic_vector( p_output_data_bits_I0 – 1 DOWNTO 0);
    );
END hive_fu_rtfpsu;

ARCHITECTURE rtl OF hive_fu_rtfpsu IS
    CONSTANT _p_max_output_delay : INTEGER := 0;
    BEGIN
    ...
    decode:PROCESS(in_optype, in_input_data) ...
    VARIABLE var_optype : INTEGER;
    VARIABLE var_output_data_I0 : std_logic_vector( p_output_data_bits_I0 – 1 DOWNTO 0);
    BEGIN
    ...
        var_optype := to_integer(unsigned(in_optype));
        CASE var_optype IS ...
            WHEN 2 | 3 => ...
        WHEN OTHERS => ...
        END CASE;
        wire_output_data_I0 <= var_output_data_I0;
    END PROCESS;
    ...
END rtl;
AVISPA instance example
accelerator for software defined radio
- 16 bit datapath
- 41 issue slots, 75 function units
- 95 register files
- 5 dual port data mems
- 115K gates excl. memories
- 150 MHz (standard cell, CMOS12, WCML)
- 0.85 mW/MHz

Moustique instance example
accelerator for mobile video coding
- 32 bit SIMD datapath
- 4 issue slots, 25 function units, 11 register files
- 1 single port data memory, 2-way set-assoc. dcache
- 84K gates excl. data/program mems
- 100 MHz (standard cell, CMOS12, WCML)
Summary

• highly scalable computationally efficient architecture template
• allows massive instruction-level parallelism, e.g. revised AVISPA can issue up to 60 operations per cycle
• processor design methodology enables finding proper match between architecture, compiler, and application domain
• wide range of high-level programmable cores automatically generated
• cores aimed at replacing traditional signal-processing accelerators (ASICs, DSPs) in SoCs
• cores enable high performance, low power, high-level programmable SoCs

www.siliconhive.com