TOKYO TECH
Pursuing Excellence

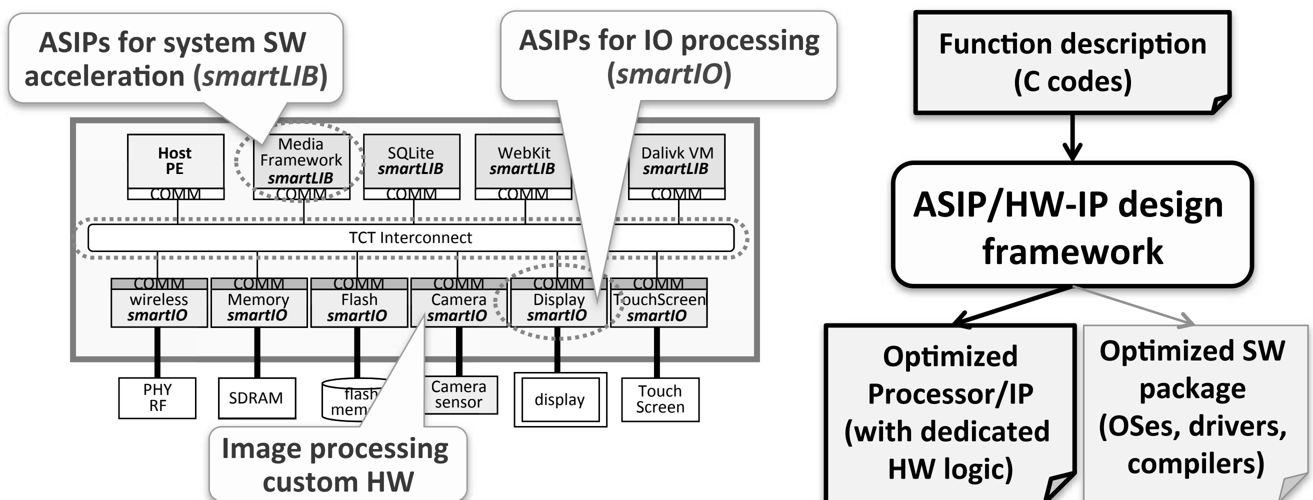
C-Based RTL Design Framework for Hardware-IP and ASIP Synthesis

Tsuyoshi Isshiki

Global Scientific Information and Computing Center
Dept. of Communications and Computer Engineering
Tokyo Institute of Technology

MPSoC '15
July. 15th, 2015

Heterogeneous Multicore SoC Design Challenges

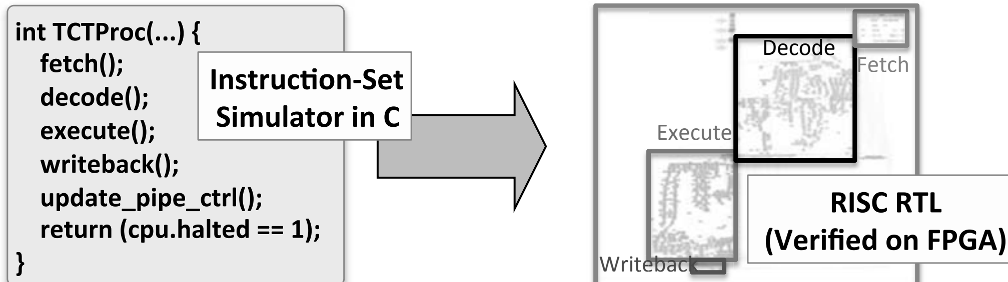


- **Functionally distributed system architecture**
  - **Distribute complex system functionalities to different functional blocks**
  - **Implement each function on optimized ASIP & custom HWs**

→ ***How to efficiently design these large # of ASIPs and custom HWs??***

Proposed C-Based HW-IP & ASIP Synthesis Framework

- **Direct RTL implementation from C descriptions**
  - **Data-flow style descriptions on C → direct circuit mapping**
  - **Pragma-based HW attribute annotations: bit-width, register, memory**
- **Design cases**
  - **Image processing algorithms in C → high throughput pixel pipeline**
  - **Instruction-set simulator in C → RISC processor RTL** (*including CACHES, MMU, IO peripherals such as UART: all synthesizable to RTL*)



For SW developers to write *"their own processors"*

Comparison with Conventional EDA Tools

- **High-Level Synthesis: SW description → RTL synthesis**
  - ✧ **Behavioral synthesis via parallelizing compiler technology**
  - ✧ **Synthesized RTL architecture heavily dependent on HLS tools (need to follow tool-specific coding styles)**

- **Processor design tools: processor description → RTL synthesis**
  - ✧ **ISA/micro-architecture description → ISS / compiler/ RTL generation**
  - ✧ **Proprietary language (hard to learn), limited RTL synthesis capability**

- **Proposed C description format & synthesis technology:**
  - ✧ **Direct RTL description using "C-dataflow" coding style**
  - ✧ **Fully ANSI-C compliant → can use any C development tool**
  - ✧ **Pragma-based hardware attributes: bit-widths, clock boundaries**
  - ✧ **Naturally describes pipelining, parallelisms, state machines**
  - ✧ **Real design cases on image processing and processor designs**

RISC Processor Resource Description

```
typedef unsigned char      UINT8;
typedef unsigned int       UINT32;
typedef UINT8              BIT, UINT2, UINT4;
typedef BIT                ST_BIT;
typedef UINT32             M_UINT32, ST_UINT32;

typedef struct {
  BIT        stall, ready, stalled;
  ST_BIT     stall_fw;
} PipeCtrl;

typedef struct { PipeCtrl pctl;} FEState;

typedef struct { UINT32  cur_pc, nxt_pc; } FEReg;

#pragma _TCT_verilog_bit_width      1  BIT
#pragma _TCT_verilog_bit_width      2  UINT2
#pragma _TCT_verilog_bit_width      4  UINT4
#pragma _TCT_verilog_memory            M_UINT32
#pragma _TCT_verilog_state             ST_BIT ST_UINT32
#pragma _TCT_verilog_state    FEReg DCReg EXReg DIVReg
```

```
typedef struct ST_CPU
{
  ST_UINT32 gpr[GPR_COUNT];
  ST_UINT32 spr[SPR_COUNT];
  M_UINT32  pmem[PM_SIZE];
  M_UINT32  dmem[DM_SIZE];
  ST_BIT    halted;
  ST_UINT32 cycle, ir_prev;
  UINT32    ir;
  Insn      insn;
  FEState   fe_stt;
  DCState   dc_stt;
  EXState   ex_stt;
  WBState   wb_stt;
  FEReg     fe_reg;
  DCReg     dc_reg;
  EXReg     ex_reg;
  DIVReg    div;
  IO        io;
  Exception except;
} CPU;
extern CPU cpu;
```

Register files

memories

- No built-in data types
- No C language extensions
- → *Only C data types*
- → *Can execute on standard C development platforms*

Pipeline registers

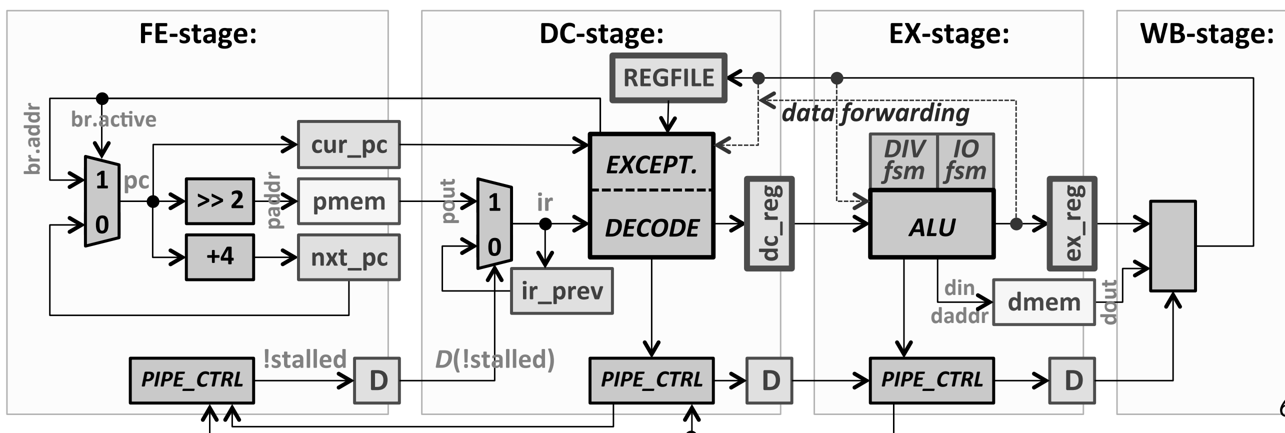
- Pragma attributes on C-typedefs
- *state variables → registers*
- *non-state variables → wires*

Single Clock Cycle Behavior Description

```
int TCTProcUART(BIT *uart_rx, DO_BIT *uart_tx)
{
  set_uart_ports(uart_rx, uart_tx);
  fetch();
  decode();
  execute();
  writeback();
  cpu.cycle ++;
  return (cpu.halted == 1);
}
```

Top-level C-model function

- C-coding semantics: One call to the top-level function models a *single cycle behavior* of the total system
- CODING RULE: *each register variable and memory variable should be assigned at most once during the top-level function call*

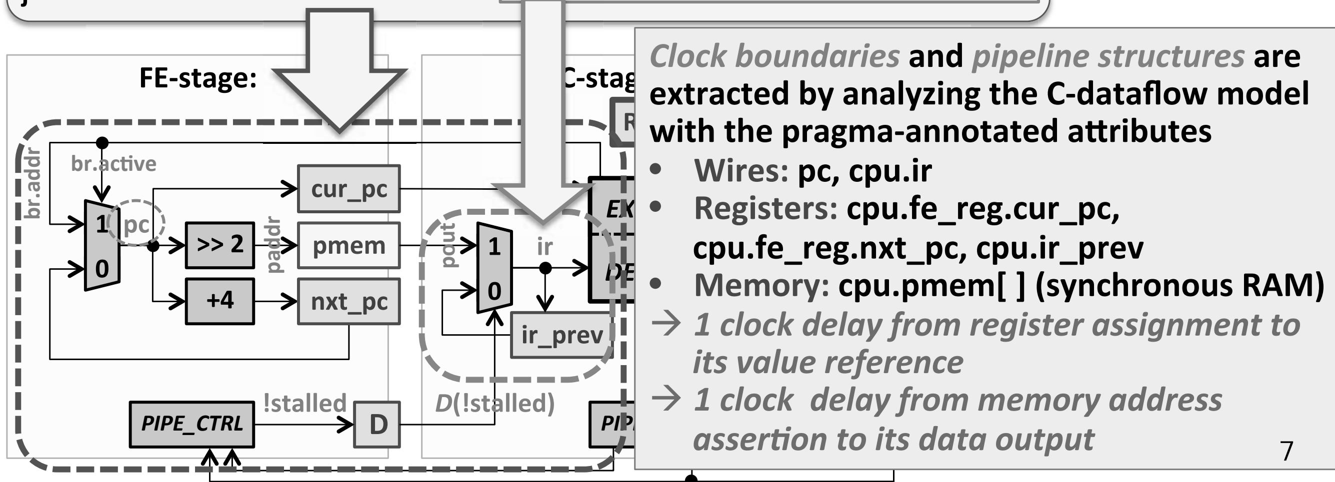


C-Dataflow Model Coding Style

```
void fetch() {
  cpu.fe_stt.pctl.stalled = cpu.dc_stt.pctl.stall | cpu.ex_stt.pctl.stall;
  if(!cpu.fe_stt.pctl.stalled){
    UINT32 pc = (cpu.dc_stt.br.active) ? cpu.dc_stt.br.addr : cpu.fe_reg.nxt_pc;
    cpu.fe_reg.cur_pc = pc;
    cpu.ir = cpu.pmem[pc >> 2];
    cpu.ir_prev = cpu.ir;
    cpu.fe_reg.nxt_pc = pc + 4;
  } else{ cpu.ir = cpu.ir_prev; }
}
```

Directly captures the dataflow between registers, memories and wires on the C semantics

assignments delayed by 1 clock from "pc" signal in the RTL model



Clock boundaries and *pipeline structures* are extracted by analyzing the C-dataflow model with the pragma-annotated attributes

- Wires: pc, cpu.ir
- Registers: cpu.fe_reg.cur_pc, cpu.fe_reg.nxt_pc, cpu.ir_prev
- Memory: cpu.pmem[ ] (synchronous RAM)

→ *1 clock delay from register assignment to its value reference*

→ *1 clock delay from memory address assertion to its data output*

Backward Signal Propagation

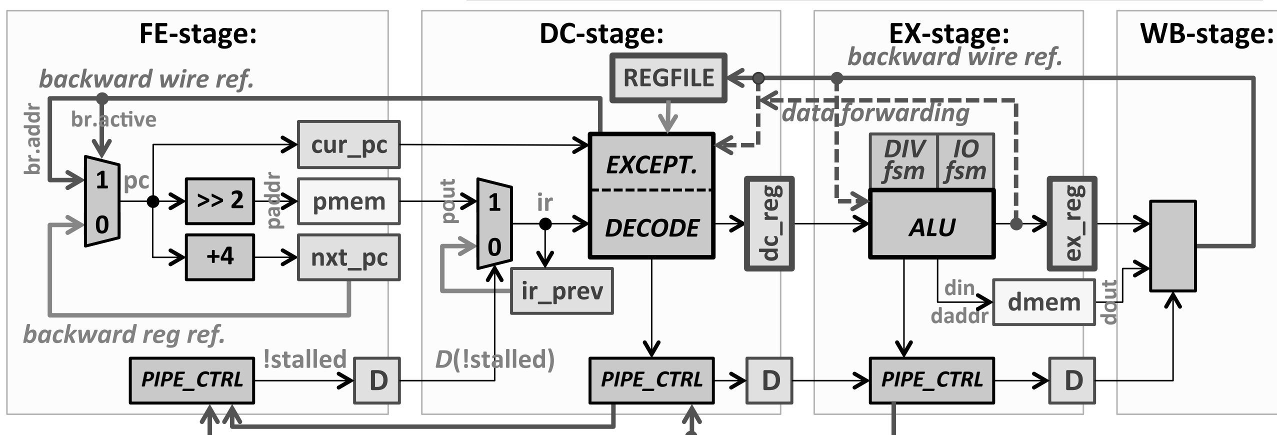
```
void fetch() {
  cpu.fe_stt.pctl.stalled = cpu.dc_stt.pctl.stall | cpu.ex_stt.pctl.stall;
  if(!cpu.fe_stt.pctl.stalled){
    UINT32 pc = (cpu.dc_stt.br.active) ? cpu.dc_stt.br.addr : cpu.fe_reg.nxt_pc;
    cpu.fe_reg.cur_pc = pc;
    cpu.ir = cpu.pmem[pc >> 2];
    cpu.ir_prev = cpu.ir;
    cpu.fe_reg.nxt_pc = pc + 4;
  } else{ cpu.ir = cpu.ir_prev; }
}
```

backward wire reference

backward register reference

Value reference whose assignment is *not reachable* on the C-dataflow model → *backward reference*

- *Backward register reference*: value assignment occurs on the same pipe-stage ("current state")
- *Backward wire reference*: value assignment occurs on the later pipe-stages



Cache Modeling

```
typedef struct ST_DCache
{
  DCacheState stt;
  DCacheReg   reg;
  M_UINT32    tag   [DWAYS][DLINES];
  M_UINT32    word [DWAYS][DLINES][DWORDS];
} DCache;
```

Multi-dimensional memory array
- tag[#ways][#lines]
- word[#ways][#lines][#words]

```
UINT32 read_dcache_line(int way, int lidx, int widx)
{   return cpu.dcache.word[way][lidx][widx]; }
```

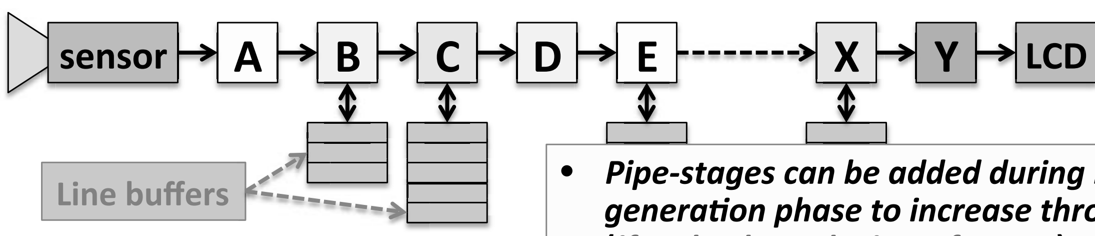
```
BIT read_dcache_tag(int lidx, UINT32 tag)
{
  int       w;
  BIT       hit_flag  = 0, empty_flag = 0;
  UINT4     hit_way = 0, empty_way= 0;
  UINT4     flush_way = cpu.dcache.reg.flush_way;
  UINT32    hit_tag = 0, flush_tag     = 0;

  for(w = 0; w < DWAYS; w ++){
    UINT32 ctag          = cpu.dcache.tag[ w ][ lidx ];
    BIT is_hit           = ((ctag >> 1) == (tag >> 1));
    BIT is_empty         = (ctag & 2) == 0;
    hit_flag             = is_hit       |  hit_flag;
    empty_flag           = is_empty   |  empty_flag;
    hit_way              = (is_hit)      ?    w      :   hit_way;
    empty_way            = (is_empty) ?    w      :   empty_way;
    hit_tag              = (is_hit)      ?    ctag   :   hit_tag;
    flush_tag            = (j == flush_way) ? ctag :  flush_tag;
  }
  cpu.dcache.stt.way      = (hit_flag) ? hit_way : (empty_flag) ? empty_way : flush_way;
  cpu.dcache.stt.tag      = (hit_flag) ? hit_tag   : (empty_flag) ? 0              : flush_tag;
  cpu.dcache.reg.flush_way  = (flush_way + 1) & DWAY_MASK; // simple round-robin
  return  hit_flag;
}
```

- ***Loops are <u>completely unrolled</u>***
- ***Wires with loop-carried dependence are merged with combinational logic***

hit_flag : ORed
hit_way, hit_tag: MUXed

Camera Image Signal Processing C Description



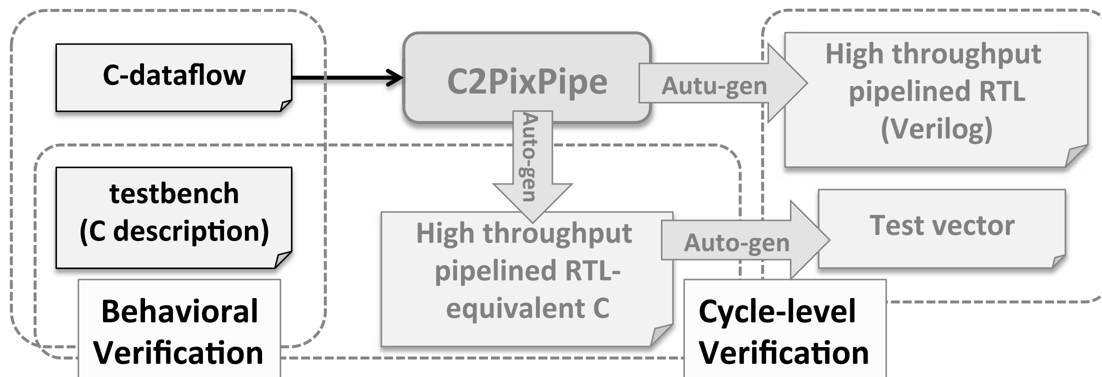
- ***Pipe-stages can be added during RTL generation phase to increase throughput (if no backward wire reference)***

ISP block C functions (10 ISP blocks → 32 pipe-stages)

```
void IpxTop(          IpxTopCtx* ctx,
                      const XSVI_UINTXX* xsviIn,     XSVI_RGB08* xsviOut,
                      const IpxSVIParam* sp,          const IpxTopParam* tp)
{
    Xsvi2Ipx_UINTXX     (&ctx->Vif, xsviIn, &svi0, &data0, sp);
    IpxOBC     (&ctx->OBC, &svi0, &svi1, &data0, &data1, NULL, &tp->pOBC);
    IpxWBG     (&ctx->WBG, &svi1, &svi2, &data1, &data2, NULL, &tp->pWBG);
    IpxBPF     (&ctx->BPF, &svi2, &svi3, &data2, &data3, sp, &tp->pBPF);
    IpxWNR     (&ctx->WBR, &svi3, &svi4, &data3, &data4, sp, &tp->pWBR);
    IpxACI     (&ctx->ACI, &svi4, &svi4, &data4, &data5, sp, &tp->pACI);
    IpxCCM     (&ctx->CCM, &svi5, &svi5, &data5, &data6, NULL, &tp->pCCM);
    IpxYCN     (&ctx->YCN, &svi6, &svi6, &data6, &data7, sp, &tp->pYECN);
    IpxGAM     (&ctx->GAM, &svi7, &svi8, &data7, &data8, NULL, &tp->pGAM);
    Ipx2Xsvi_RGB08       (&svi8, &data8, xsviOut, sp);
}
```

Proposed C-Based System Designs Framework

- C-dataflow model : single cycle behavior model of the total system
  - Pragma annotations for HW attributes (bit-width / register / memory)
  - Single cycle restriction: *cannot assign registers/memories more than once*
  - Backward reference: describe *FSMs, pipeline controls, data forwarding*
  - Complete unrolling of loops and function calls
- Tool features for RTL generation
  - Bit-width optimization on internal signals (wires/registers)
  - User-specified pipe-stage count (if no backward wire reference)
  - Pipeline boundary optimization (register retiming) for maximum clock frequency
  - Generates RTL-equivalent C model for facilitating RTL verification



RISC Processor C2RTL Generation Results

RISC features: 32-bit data, 4 stage pipeline, 32 general purpose registers, integer multiply unit, multi-cycle integer division unit, UART port, interrupt handling logic	
C code size	1,071 lines, 4 source files
# operations	2,595 ops (1 multiplication)
# pipeline stages	4
RTL code size	4,100 lines (Verilog), 3,650 lines (C)
gate count	72,986 gates (approx.)
FF count	4,436 bits (pipeline registers + reg-file)
max clock freq.	300 MHz (approx., @90nm CMOS)
RTL synthesis time	19.1 seconds

test program: calculation of 200 prime numbers with UART message of output results (5,600 characters @ 115.2K bps)		
simulation cycles	10,764,342 cycles	
ISS time (C dataflow)	0.918 sec	11.726 M cycles/sec
ISS time (RTL-equiv. C)	5.558 sec	1.937 M cycles/sec
ISS time (Verilog)	38.850 sec	0.277 M cycles/sec

Camera Image Signal Processing C2RTL Generation Results

ISP functions: Pixel interface conversion, color adjustments, Bayer-RGB demosaic, denoising, color conversion, gamma correction, etc.	
C code size	5,000 lines, 15 source files
# operations	4,195 ops (76 multiplications)
line buffer memory	64 lines (2,304 pixels wide), 1.9M bits
# pipeline stages	32
RTL code size	27,000 lines (Verilog), 20,000 lines (C)
comb. circuit size	225,279 gates (approx.)
FF count	23,393 bits
max clock freq.	300 MHz (approx., @90nm CMOS)
RTL synthesis time	56.8 seconds (excl. C parsing)

Summary

- C-dataflow model
  - No language extensions, no built-in functions or structures → *runs on any C development platforms*
  - Single cycle behavior model of the entire system
  - Directly captures the dataflow between registers, memories and wires
  - HW attributes (bit-width, registers, memories) annotated by pragmas
  - Easily describes *FSMs, pipeline controls* and *data-forwarding* using *backward references*
  - Easily describes memory subsystems (caches, MMUs) and IOs (pin-accurate UART)
- RTL generation
  - Bit-width optimization on internal signals
  - Pipeline boundary optimization for maximum clock frequency operation
  - RTL-equivalent C generation for efficient RTL verification
- On-going
  - Linux porting on our C-based processor model
  - Image processing ASIPs and custom HWs

Thank You for Your Attention!

Tsuyoshi Isshiki

isshiki@vlsi.ce.titech.ac.jp

Global Scientific Information and Computing Center

Dept. Communications and Computer Engineering

Tokyo Institute of Technology