ASIP Design Methodology on C2RTL Framework

Tsuyoshi Isshiki

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

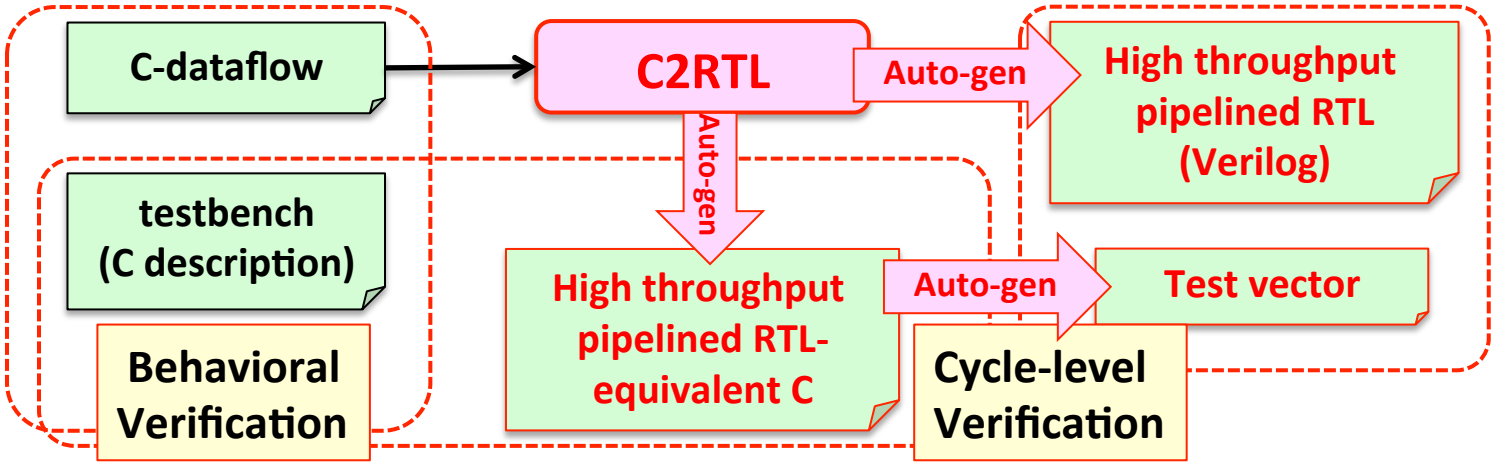
MPSoC ‘16
July. 11th, 2016

C2RTL Design Framework [MPSoC '15]

- **C-dataflow model : single-cycle behavior model of the total system**
  - Single-cycle restriction: *cannot assign registers/memories more than once*
  - Backward reference: describe *FSMs, pipeline controls, data forwarding*
  - Pragma annotations for HW attributes (bit-width / register / memory)

  → ***Any RTL structure can be precisely described with C-dataflow model***
- **C2RTL tool features**
  - 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

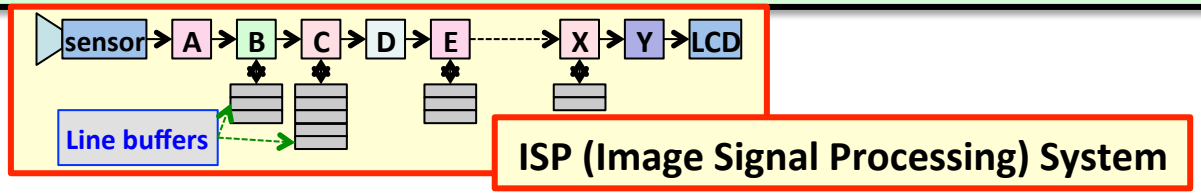
Japan Patent : No.5842255 (2015) (PCT application pending)



C-Dataflow Coding Styles

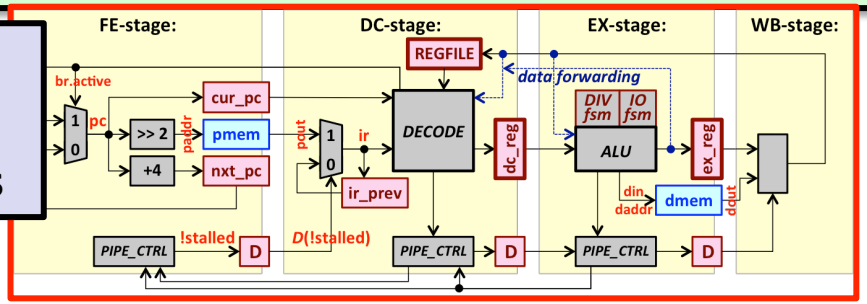
- **Simplistic dataflow pipeline coding style**
  - ✧ High parallelism & pipelining → TeraOPS throughput RTL arch.
  - ✧ No pipeline stalls → straightforward C dataflow coding
  - ✧ # pipe stages, register retiming (for max freq.) → fully automated

▪ signal processing



- **Flow-controlled complex pipeline coding style**
  - ✧ Complex flow-control between pipe-stages (stalls, data forwarding)
  - ✧ Detailed microarchitecture description using C dataflow coding
  - ✧ All kinds of interface logic, memory controllers, busses, NoCs

▪ processors
▪ controls
▪ communications



RISC Processor Resource Description [MPSoC '15]

```
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

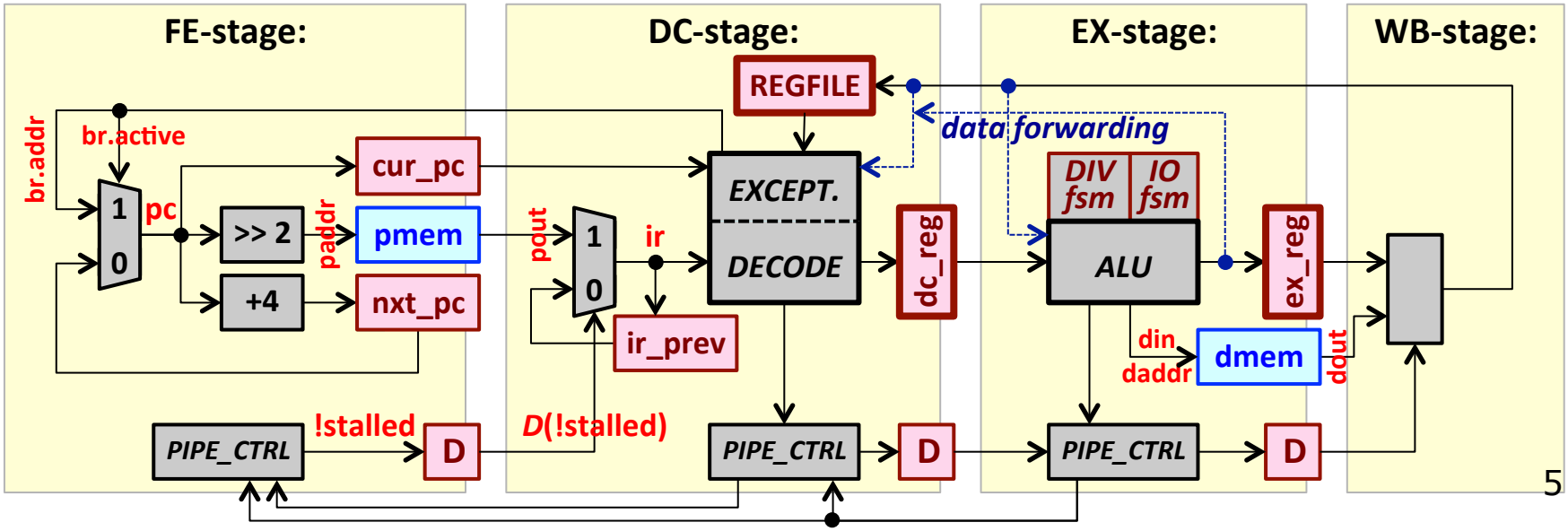
- Pragma attributes on C-typedefs
- *state variables → registers*
- *non-state variables → wires*

Single-Cycle Behavior Description [MPSoC '15]

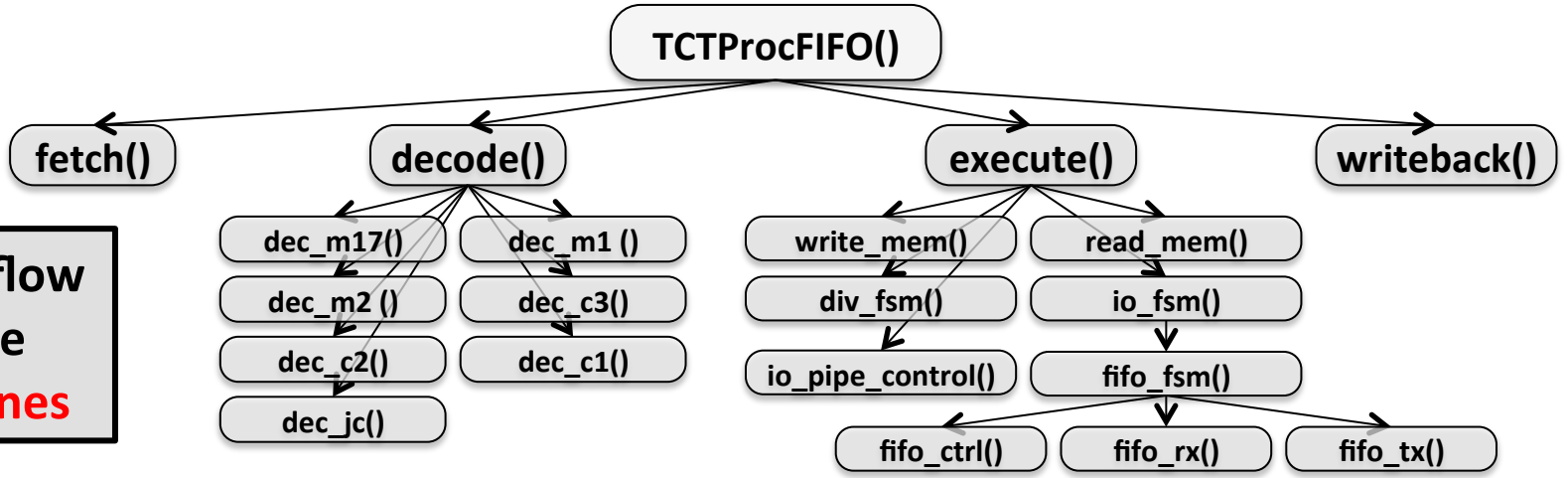
```
int TCTProcUART(BIT *uart_rx, DO_BIT *uart_tx)
{
  set_uart_ports(uart_rx, uart_tx);
  fetch();
  decode();
  execute();
  writeback();
  cpu.cycle ++;
  return (cpu.halted == 1);
}
```

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 and RTL Generation



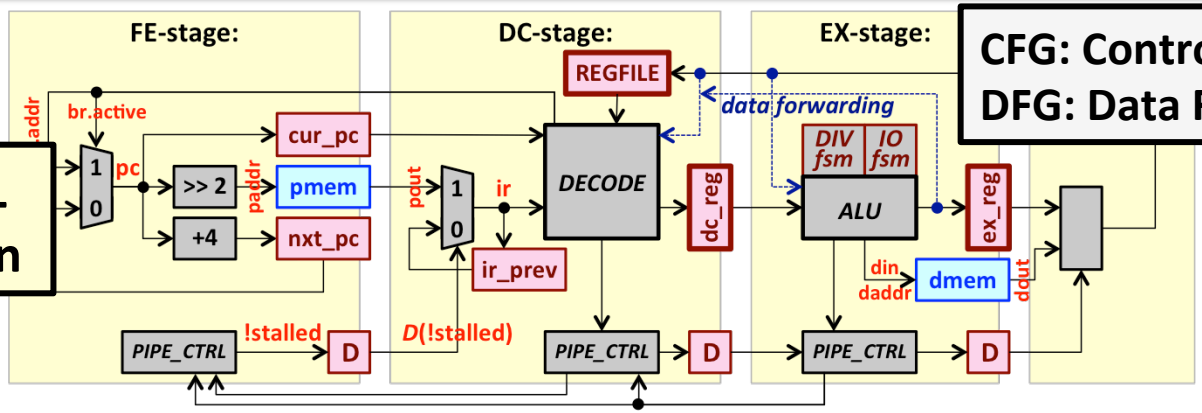
C dataflow code
440 lines

- Full function inlining, full loop unrolling
- CFG → DFG degeneration
- DFG → RTL conversion (RTL structure fully described on C)

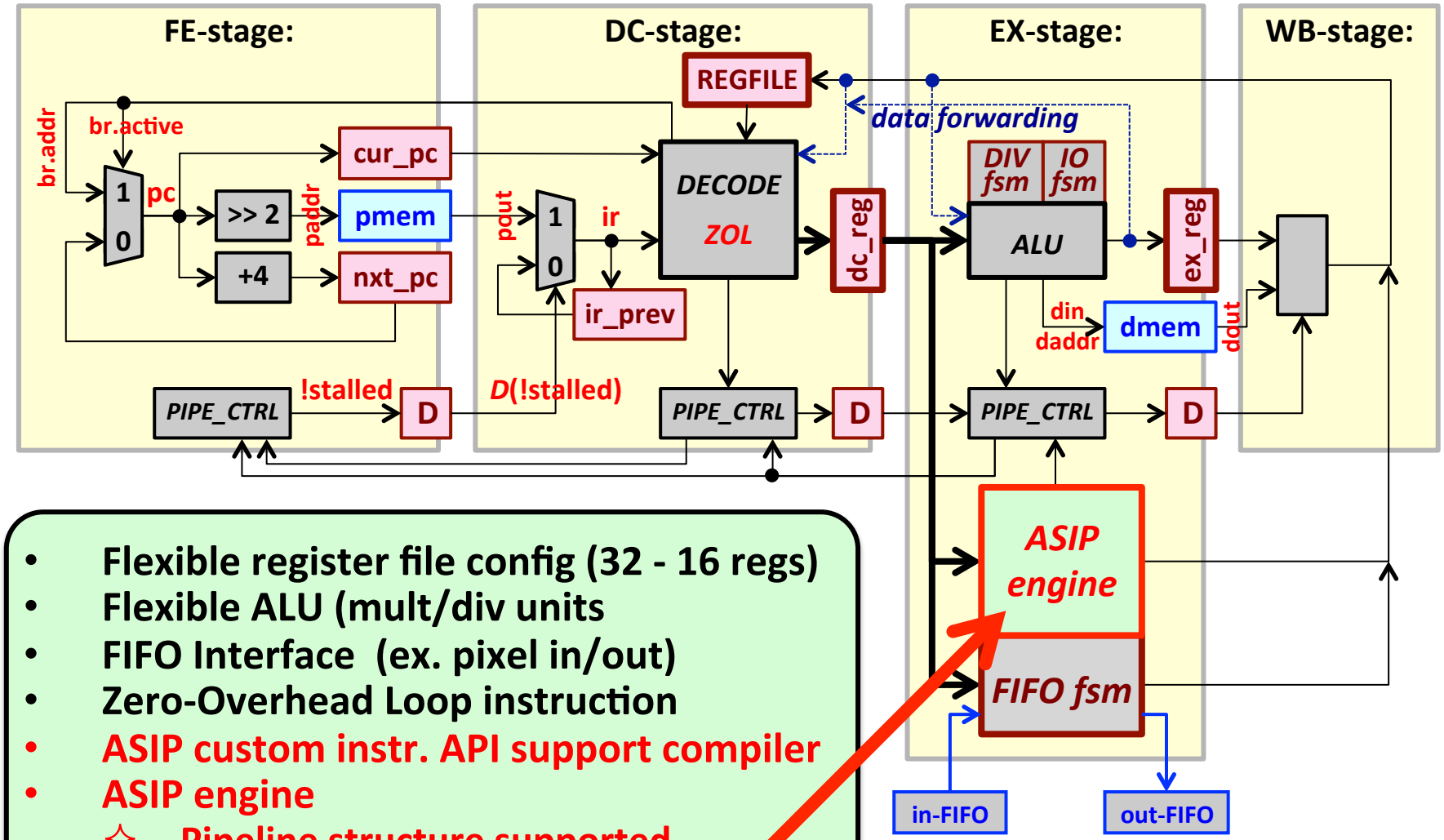
Made possible by *single cycle behavior* semantic

CFG: Control Flow Graph
DFG: Data Flow Graph

Direct RTL conversion



C2RTL ASIP Framework



- Flexible register file config (32 - 16 regs)
- Flexible ALU (mult/div units
- FIFO Interface (ex. pixel in/out)
- Zero-Overhead Loop instruction
- ASIP custom instr. API support compiler
- ASIP engine
  - ✧ Pipeline structure supported
  - ✧ Dedicated register/memory

JPEG Encoder ASIP Design Case

- **JPEG encoder SW reference**
  - ✧ Hot spots: color conv. (4:2:0), 8-point DCT, quant, Huffman coding
  - ✧ SW exe. cycles: 383,323,249 clks @1600x1200 pixels
  - ✧ TCT processor area: 42,534 gates (C2RTL estimate)

199.65 clks/pixel

59.48x speedup !!!

3.356 clks/pixel

- **ASIP design**

Design effort: 2 weeks x 1 person!!

  - ✧ Custom instruction designs on hot spot functions
  - ✧ ASIP exe. cycles: 6,444,351 clks @1600x1200 pixels
  - ✧ ASIP area (incl. proc core) : 111,634 gates (area ratio: 2.62x)

Comparable with commercial JPEG encoder IP (area, performance) !!

custom Instr.	throughput	cycle reduction	area
color conv.(4:2:0)	1 clk/pixel (RGB)	132,040,132 clks (34.4%)	11,629 gates
8-point DCT	16 clks/8x8-block	84,679,499 clks (22.1%)	48,977 gates
quantization	~1 clk/DCT coef	76,728,540 clks (20.0%)	7,097 gates
Huffman coding	~1 clk/code word	83,430,727 clks (21.8%)	5,823 gates
TOTAL		376,878,898 clks (98.3%)	73,526 gates

JPEG Encoder ASIP Custom Instructions

- **Color conversion (4:2:0) : 1 clk/pixel (RGB) → 11,629 gates**
  - ✧ RGB → YCbCr : 3x3 matrix multiplication, shift+rounding
  - ✧ CbCr downsampling(4:4:4) → (4:2:0)
  - ✧ YCbCr memory → 8-pixel wide read for 8-point DCT
- **8-point DCT : 16 clks/8x8-block → 48,977 gates**
  - ✧ 12 multiply + 32 add → 1clk operation
  - ✧ Resource sharing of H/V direction DCT processes
  - ✧ Dedicated registers → 64 x 14 bits for zig-zag scanning
- **Quantization : ～1 clk/DCT-coef → 7,097 gates**
  - ✧ 1 cycle computation for small quotients (0～7 : covers 98.3%)
  - ✧ Multi-bit "long division" FSM for large quotients (>= 8 : 1.7%)
  - ✧ Zig-zag scan FSM, run-length counter
- **Huffmann Coding : ～1 clk/code-word → 5,823 gates**
  - ✧ Bit-packing FSM for variable-length Huffman code generation
  - ✧ FIFO output (byte wide)

8-point DCT ASIP Engine Description

SW reference code

```
void DCTcore(int * cfin, int * cfout){
 /// dir = 0 (Horizontal): cfin -> data2 (loop 8 times)
 for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
  tmp0 = cfin[0] + cfin[7];  tmp7 = cfin[0] - cfin[7];
  tmp1 = cfin[1] + cfin[6];  tmp6 = cfin[1] - cfin[6];
  tmp2 = cfin[2] + cfin[5];  tmp5 = cfin[2] - cfin[5];
  tmp3 = cfin[3] + cfin[4];  tmp4 = cfin[3] - cfin[4];

  tmp10 = tmp0 + tmp3;    tmp13 = tmp0 - tmp3;
  tmp11 = tmp1 + tmp2;    tmp12 = tmp1 - tmp2;
  tmp14 = tmp10 + tmp11;  tmp15 = tmp10 - tmp11;
  data2[0] = (tmp14 << PB1);
  data2[4] = (tmp15 << PB1);

  z1 = MULT(tmp12 + tmp13, FIX_0_541196100);
  z2 = z1 + MULT(tmp13, FIX_0_765366865);
  z3 = z1 + MULT(tmp12, - FIX_1_847759065);
  data2[2] = DESCALE(z2, CB0);
  data2[6] = DESCALE(z3, CB0);
  . . . . .
 }
 /// dir = 1 (Vertical) : data2 -> cfout (loop 8 times)
}
```

Direct reuse of SW reference code

C-dataflow code for RTL synth.

```
void jpg_dct(JPG * jpg, int dir){
  tmp0 = cfin[0] + cfin[7];  tmp7 = cfin[0] - cfin[7];
  tmp1 = cfin[1] + cfin[6];  tmp6 = cfin[1] - cfin[6];
  tmp2 = cfin[2] + cfin[5];  tmp5 = cfin[2] - cfin[5];
  tmp3 = cfin[3] + cfin[4];  tmp4 = cfin[3] - cfin[4];

  tmp10 = tmp0 + tmp3;    tmp13 = tmp0 - tmp3;
  tmp11 = tmp1 + tmp2;    tmp12 = tmp1 - tmp2;
  tmp14 = tmp10 + tmp11;  tmp15 = tmp10 - tmp11;
  jpg->dct.out_data[0] = (dir == 0) ?
              (tmp14 << PB1) : DESCALE(tmp14, PB1);
  jpg->dct.out_data[4] = (dir == 0) ?
              (tmp15 << PB1) : DESCALE(tmp15, PB1);

  z1 = MULT(tmp12 + tmp13, FIX_0_541196100);
  z2 = z1 + MULT(tmp13, FIX_0_765366865);
  z3 = z1 + MULT(tmp12, - FIX_1_847759065);
  jpg->dct.out_data[2] = (dir == 0) ?
              DESCALE(z2, CB0) : DESCALE(z2, CB1);
  jpg->dct.out_data[6] = (dir == 0) ?
              DESCALE(z3, CB0) : DESCALE(z3, CB1);
  . . . . .
}
```

Merge different scaling factors on H/V directions

8-point DCT ASIP Engine Description

C-dataflow code for RTL synth.

```
void jpg_dct(JPG * jpg, int dir){
  tmp0 = cfin[0] + cfin[7];  tmp7 = cfin[0] - cfin[7];
  tmp1 = cfin[1] + cfin[6];  tmp6 = cfin[1] - cfin[6];
  tmp2 = cfin[2] + cfin[5];  tmp5 = cfin[2] - cfin[5];
  tmp3 = cfin[3] + cfin[4];  tmp4 = cfin[3] - cfin[4];

  tmp10 = tmp0 + tmp3;    tmp13 = tmp0 - tmp3;
  tmp11 = tmp1 + tmp2;    tmp12 = tmp1 - tmp2;
  tmp14 = tmp10 + tmp11;  tmp15 = tmp10 - tmp11;
  jpg->dct.out_data[0] = (dir == 0) ?
              (tmp14 << PB1) : DESCALE(tmp14, PB1);
  jpg->dct.out_data[4] =
              (tmp15 << P

  z1 = MULT(tmp12 + tm
  z2 = z1 + MULT(tmp13
  z3 = z1 + MULT(tmp12
  jpg->dct.out_data[2] =
              DESCALE(z2
  jpg->dct.out_data[6] =
              DESCALE(z3
```

8-point DCT custom instruction

- Direct reuse of SW reference code
- 1 cycle execution of inner loop
- Merge different scaling factors on H/V
- Const-multiply → shift+add automatic conversion (by C2RTL tool)

8x8 2D DCT in 16 cycles

```
  z1 = tmp4 + tmp7;  z2 = tmp5 + tmp6;
  z3 = tmp4 + tmp6;  z4 = tmp5 + tmp7;
  z5 = MULT(z3 + z4, FIX_1_175875602);
  tmp4 = MULT(tmp4, FIX_0_298631336);  tmp5 = MULT(tmp5, FIX_2_053119869);
  tmp6 = MULT(tmp6, FIX_3_072711026);  tmp7 = MULT(tmp7, FIX_1_501321110);
  z1 = MULT(z1, - FIX_0_899976223);        z2 = MULT(z2, - FIX_2_562915447);
  z3 = MULT(z3, - FIX_1_961570560);        z4 = MULT(z4, - FIX_0_390180644);
  z3 += z5;  z4 += z5;  tmp4 += z1 + z3;  tmp5 += z2 + z4;  tmp6 += z2 + z3;  tmp7 += z1 + z4;
  jpg->dct.out_data[7] = (dir == 0) ?  DESCALE(tmp4, CB0) : DESCALE(tmp4, CB1);
  jpg->dct.out_data[5] = (dir == 0) ?  DESCALE(tmp5, CB0) : DESCALE(tmp5, CB1);
  jpg->dct.out_data[3] = (dir == 0) ?  DESCALE(tmp6, CB0) : DESCALE(tmp6, CB1);
  jpg->dct.out_data[1] = (dir == 0) ?  DESCALE(tmp7, CB0) : DESCALE(tmp7, CB1);
}
```

All in 1 cycle!!!

Quantization ASIP Engine (state-0)

```
UINT32 jpg_quant_st0(JPG * jpg, int * q_zero){
  UINT16 divisor = jpg_q_get_q_table_data(jpg);
  int  q = divisor << 2, qv = q << 1;
  SINT16 coef = jpg_dct_read(jpg);
  SINT16 abs_coef = (coef < 0) ? -coef : coef;
  int  r_coef = abs_coef + q, qc = 0;
  *q_zero = (r_coef < qv);
  if(r_coef >= qv){      /// qc > 0  (10.70%)
    if(r_coef < qv*8){
      if(r_coef < qv*4){
        if(r_coef < qv*2)        { qc = 1; }
        else                     { qc = (r_coef < qv*3) ? 2 : 3; }
      } else if(r_coef < qv*6) { qc = (r_coef < qv*5) ? 4 : 5; }
      else                     { qc = (r_coef < qv*7) ? 6 : 7; }
    } else{   /// qc >= 8 (1.70%) : setup long-division FSM
        jpg_q_setup_div(jpg, coef, abs_coef2, qval2);
  }   /// else { qc = 0  (89.30%) }
  jpg_q_set_nxt(jpg);
  return (coef < 0) ? -qc : qc;
}
```

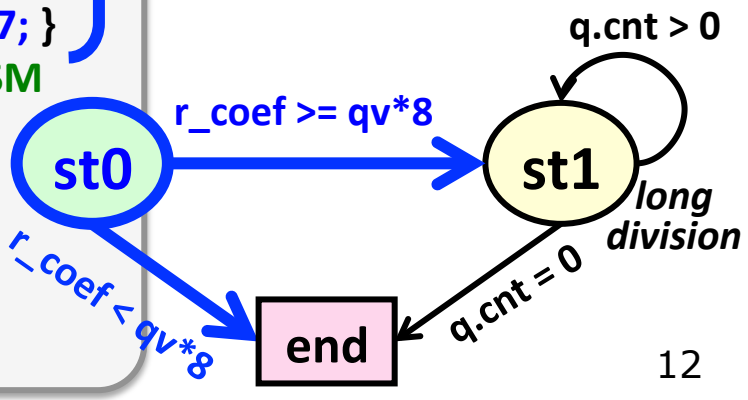
Small division (qc = r_coef / qv)

→ *98.3% are "small" quotients*

Single cycle operation | *long division*

qc = 0	qc < 4	qc < 8	qc >= 8
89.30%	97.22%	98.30%	1.70%

0 <= qc < 8 : comparisons



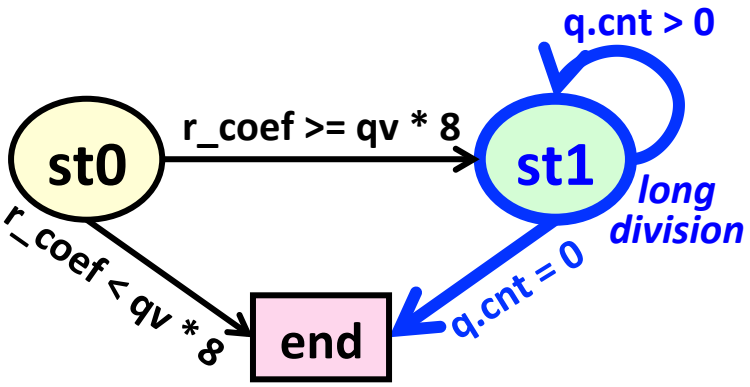
Quantization ASIP Engine (state-1)

```
void jpg_div(int * cnt_i, int * nom_i, int * rem_i,
             int * q_i, int * denom, int * cnt_o,
             int * nom_o, int * rem_o, int * q_o)
{
  int b       = (*nom_i >> 14) & 0x1;
  int nrem    = (*rem_i << 1) | b;
  int dif     = nrem - *denom;
  int qb      = (dif >= 0);
  int end     = (*cnt_i == 0);
  *rem_o      = (qb) ? dif : nrem;
  *nom_o      = (*nom_i << 1) & 0x7fff;
  *cnt_o      = (end) ? 0 : *cnt_i - 1;
  *q_o        = (end) ? *q_i : (*q_i << 1) | qb;
}
```

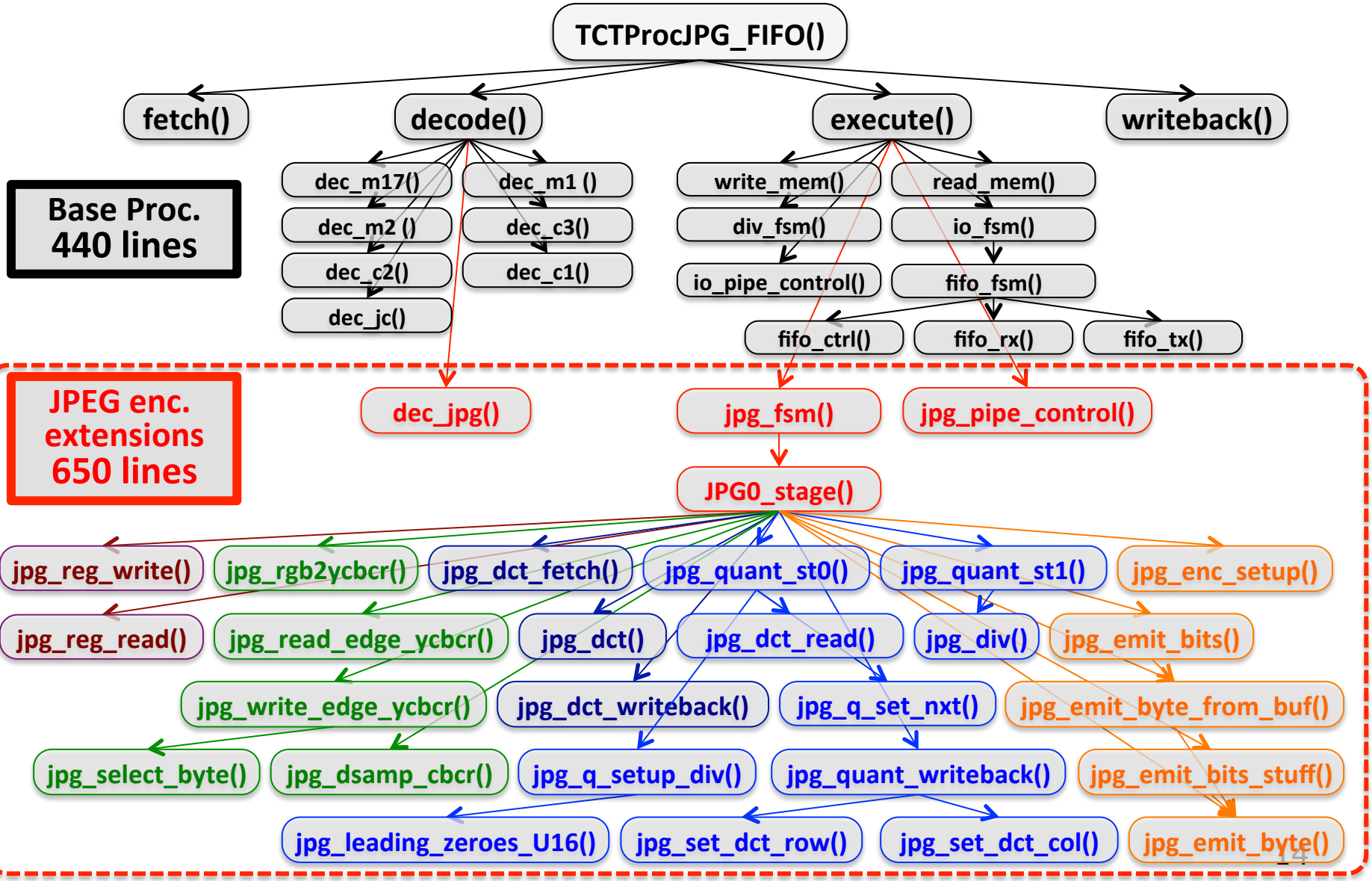
Long division FSM

DIV_ITR = 3 (process 3 bits per cycle)

```
UINT32 jpg_quant_st1(JPG * jpg)
{
#define DIV_ITR        3
  int cnt[DIV_ITR+1], nom[DIV_ITR+1];
  int rem[DIV_ITR+1], q[DIV_ITR+1];
  int denom, i;
  cnt[0]     = jpg->q.cnt;    nom[0] = jpg->q.nom;
  rem[0]     = jpg->q.rem;    q[0]   = jpg->q.q;
  denom      = jpg->q.denom;
  for(i = 0; i < DIV_ITR; i ++){  /// loop unroll (3 times)
    jpg_div(&cnt[i], &nom[i], &rem[i], &q[i], &denom,
            &cnt[i + 1], &nom[i + 1], &rem[i + 1], &q[i + 1]);
  }
  jpg->q.cnt = cnt[DIV_ITR];  jpg->q.nom = nom[DIV_ITR];
  jpg->q.rem = rem[DIV_ITR];  jpg->q.q = q[DIV_ITR];
  jpg->q.div_pending[1] = (cnt[DIV_ITR] != 0);
  return (jpg->q.neg_flag) ? -q[DIV_ITR] : q[DIV_ITR];
}
```



JPEG Encoder ASIP C-DataFlow Model



Summary

- **C2RTL Framework**
  - ANSI-C compliant → RTL verification using standard C dev. platform
  - Top function for RTL synthesis → System's single cycle behavior
  - HW attribute #pragma annotations → bit-width, registers, memories
  - Can describe all kinds of RTL structures

- **JPEG Encoder ASIP development**
  - Hot-spots : color conversion, DCT, QUANT, HUFFMAN
  - Speed-up : 59.48x → ONLY 2 weeks, 1 person effort
  - 3.356 clks/pixel、111.6k gates → comparable with commercial IP

- **ONGOING…**
  - C2RTL porting on LLVM : LLVM-IR-to-RTL !!

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