Halide-FIRRTL Design Flow

Learnings and Issues

Taesu Kim (tae.s.kim@intel.com)

Intel Labs/ADR/MRL

Mar. 2018

Agenda

- Halide-FIRRTL Design Flow
- Mapping Halide IR to Hardware
- For-loop Block Implementation
- Issues and Solutions
- Future Works

Halide FIRRTL Design Flow

- Object: To generate ASIC RTL from Halide.
- Using this flow you are able to generates FIRRTL code from Halide.
  - FIRRTL: https://bar.eecs.berkeley.edu/projects/2015-firrtl.html
- The generated FIRRTL code can be converted to Verilog-HDL using firrtl utility: https://github.com/freechipsproject/firrtl/blob/master/utils/bin/firrtl
- This flow is based on Halide/HLS: https://github.com/jingpu/Halide-HLS
- Download:

  git clone -b HLS-FIRRTL -- https://github.com/kevinkim06/Halide-HLS.git

Mapping Halide IR to Hardware

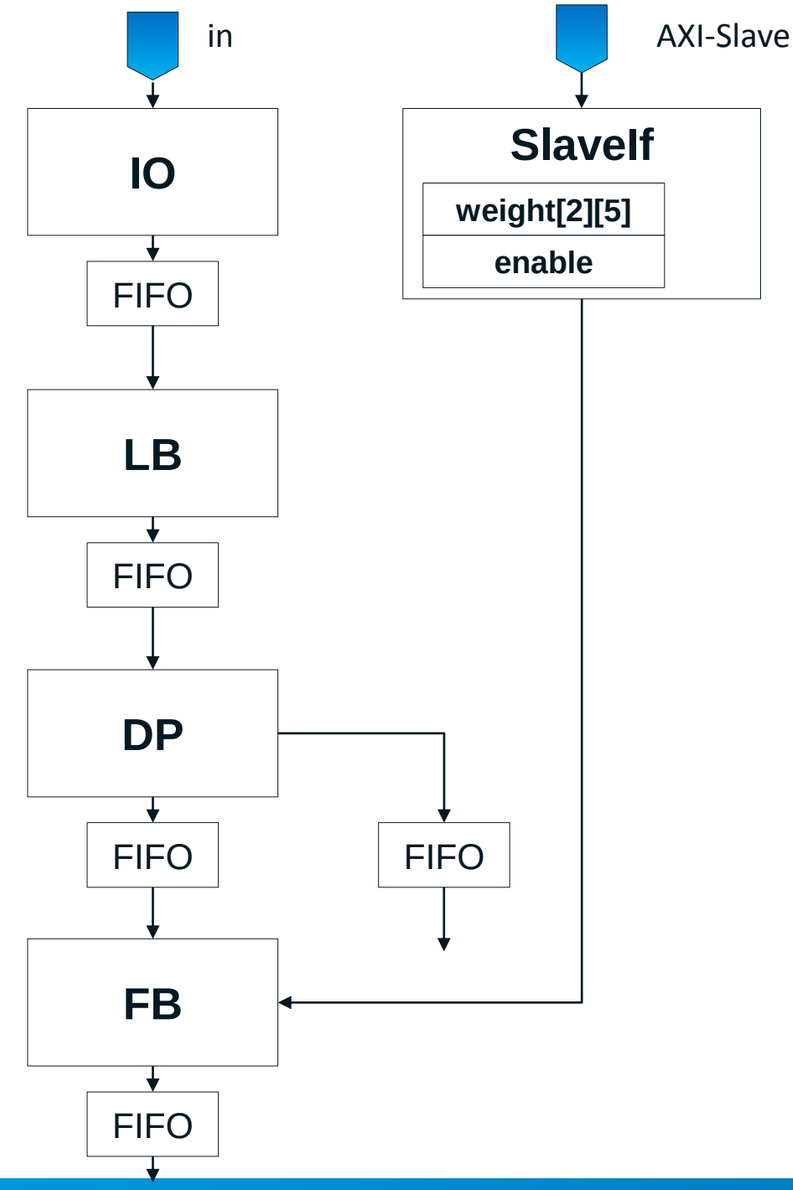
An algorithm specified in Halide Language.

```
blur_x(x, y) = (in(x, y) * weight(0,0) +  in(x+1, y) * weight(0,1) +  in(x+2, y) * weight(0,2)
          +  in(x+3, y) * weight(0,3) +  in(x+4, y) * weight(0,4))/5;
```

Added by add_kernel()

```
produce _hls_target.hw_output {
 realize in.stencil.stream([0, 5], [0, 1]) {
  linebuffer(in.stencil_update.stream, in.stencil.stream, 260, 258)
  dispatch_stream(in.stencil.stream, ..., "blur_x", ..., "hw_output", ...)
  realize blur_x.stencil_update.stream([0, 1], [0, 1]) {
   produce blur_x.stencil_update.stream {
    for (blur_x.y.__scan_dim_1, 0, 258) {
     for (blur_x.x.__scan_dim_0, 0, 256) {
      realize in.stencil([0, 5], [0, 1]) {
       read_stream(in.stencil.stream, in.stencil, "blur_x")
       realize blur_x.stencil([0, 1], [0, 1]) {
        blur_x.stencil(0, 0) = ((((((in.stencil(0, 0)*weight.tap.stencil(0, 0)) + (in.stencil(1,
0)*weight.tap.stencil(0, 1))) + (in.stencil(2, 0)*weight.tap.stencil(0, 2))) + (in.stencil(3,
0)*weight.tap.stencil(0, 3))) + (in.stencil(4, 0)*weight.tap.stencil(0, 4)))/(uint8)5)
        write_stream(blur_x.stencil_update.stream, blur_x.stencil)
       }
      }
     }
    }
   }
  }
```

Data flow type Halide IR converted by Halide/HLS.



Types of For-Loop Block (FB)

Type 1) Without Stencil Loop

```
for (y, ...) {                          <- Scan Loop
  for (x, ...) {
    realize in.stencil(...) {
      read_stream(in.stencil.stream, in.stencil, ...)
      realize blur_x.stencil(...) {
        blur_x.stencil(0, 0) = in.stencil(0,0) ...
        write_stream(blur_x.stencil.stream, blur_x.stencil)
      }
    }
}
```

- Scan loop iteration depends on input validaty AND output readiness.

Type 2) With Stencil Loop

```
for (y, ...) {                          <- Scan Loop
  for (x, ...) {
    realize in.stencil(...) {
      realize blur_x.stencil(...) {
        for (c, 0, 3) {                 <- Stencil Loop
          if (c==0) {
            read_stream(in.stencil.stream, in.stencil, ...)
          }
          blur_x.stencil(0, 0, c) = in.stencil(0,0,0) ...
          if (c==2) {
            write_stream(blur_x.stencil.stream, blur_x.stencil)
          }
        }
      }
    }
}
```

- Unlike scan loop, stencil loop can iterate freely.
- No flow control required for stencil loop.
- Detecting Stencil loop: Does not contain **realize**.

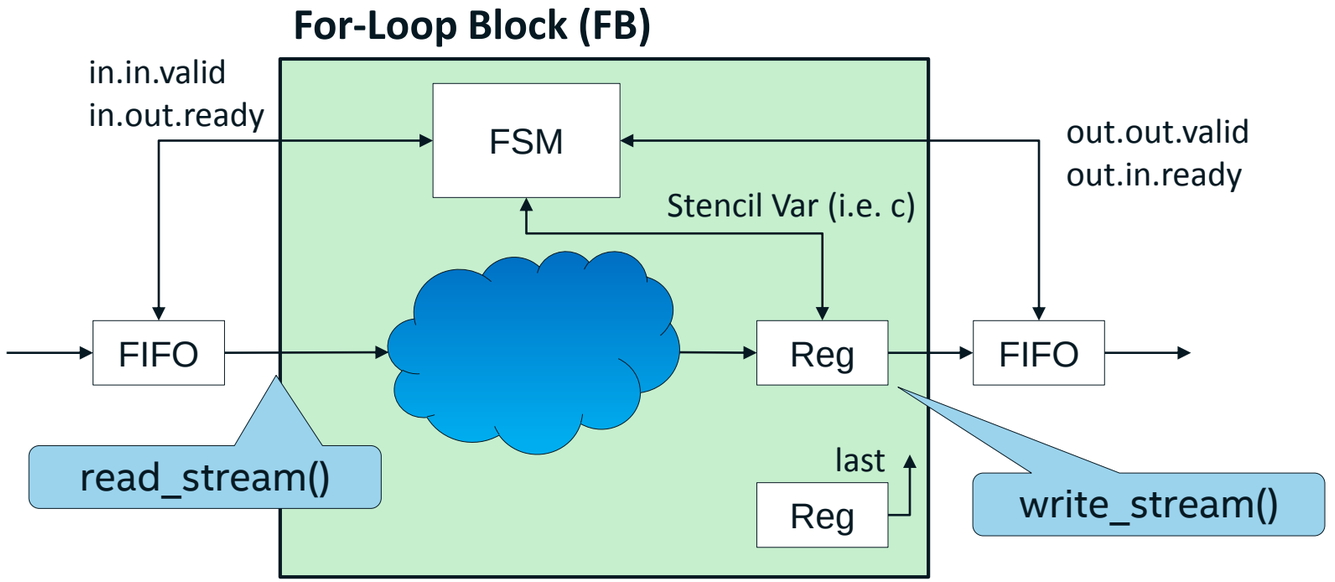
Types of For-Loop Block (cont.)

Type 3) Without Perfect Nested Loop

```
for (y, ...) {                          Scan Loop
  for (x, ...) {
    realize in.stencil(...) {
      produce in.stencil {
        read_stream(in.stencil.stream, in.stencil, ...)
      }
      realize blur_x.stencil(...) {
        produce blur_x.stencil {
          for (c, 0, 3) {                ——— Stencil Loop
            blur_x.stencil(0, 0, c) = in.stencil(0,0,0) ...
          }
        }
        write_stream(blur_x.stencil.stream, blur_x.stencil)
      }
    }
  }
}
```

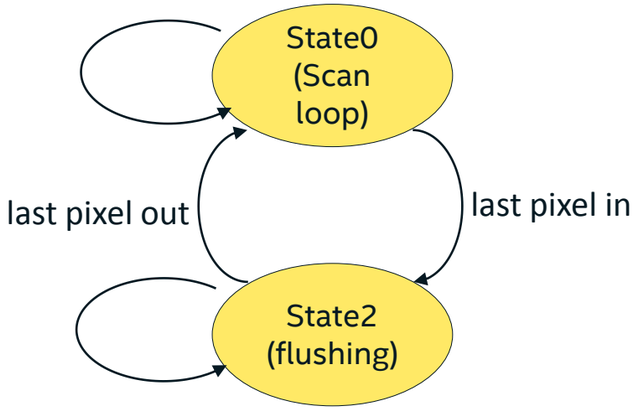
- Perfect Nested Loop is enabled by default in Halide/HLS.
- FIRRTL generation without Perfect Nested Loop is also supported.
- Same, Stencil loop doesn't contain **realize**.

FSM for For-Loop Block

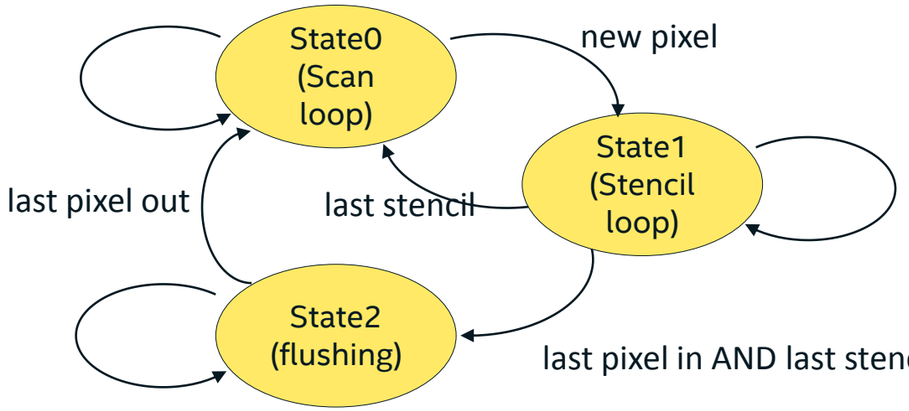


Single stage pipeline for now.

Type 1 and Type 2/3 when Stencil Loop extent is 1.



Type 2/3 when Stencil Loop extent is >1.



Example FIRRTL Code for For-Loop Block

```
for (y, ...) {
  for (x, ...) {
    realize in.stencil(...) {
      realize blur_x.stencil(...) {
        for (c, 0, 3) {
          if (c==0) {
            read_stream(in.stencil.stream, in.stencil, ...)
          }
          blur_x.stencil(0, 0, c) = in.stencil(0,0,0) ...
          if (c==2) {
            write_stream(blur_x.stencil.stream, blur_x.stencil)
          }
        }
      }
    }
  }
}
```

Halide IR

```
when started :
  when blur_x.ready :
    when eq(state, UInt<2>(0)) :
...
    when eq(state, UInt<2>(1)) :
...
    when eq(state, UInt<2>(2)) :
...
when run_step :
  in_stencil <= in.value
  node _426 = weight_tap_stencil[4][0]
  node _427 = in_stencil[asUInt(c)][0][4]
  node _428 = bits(mul(_427, _426), 7, 0)
  node _429 = weight_tap_stencil[3][0]
  node _430 = in_stencil[asUInt(c)][0][3]
  node _431 = bits(mul(_430, _429), 7, 0)
...
  node _442 = tail(add(_441, _434), 1)
  node _443 = tail(add(_442, _431), 1)
  node _444 = tail(add(_443, _428), 1)
  node _445 = UInt<8>(5)
  node _446 = div(_444, _445)
  blur_x_stencil[asUInt(c)][0][0] <= _446
  node _447 = SInt<32>(2)
  node _448 = eq(c, _447)
  when _448 :
    blur_x_stream.value <= blur_x_stencil
```

FSM

FSM is generated based on the template and collected information (scan/stencil variable name, min, max).

read_stream()

Optimization: in.value will stay constant from previous FIFO until it is popped. There is no action required for read_stream().

Compute Stage

Actual body of the For-Loop block. **Currently scheduled in a single stage pipeline.**

IfThenElse

write_stream()

Generated FIRRTL

Issue – Type Casting (Bool to Int)

In Halide:

```
hw_output(x, y) = (in(x, y) > threshold) * in(x, y); // Boolean type is used in math.
```

In the generated HLS C:

```
_in_stencil = _in_stencil_stream.read();
int8_t _240 = _in_stencil(0, 0);
int32_t _241 = (int32_t)(_240);
bool _242 = _threshold < _241;
int8_t _243 = (int8_t) (_242); // _243 is either 0 or 1
int8_t _244 = _243 * _240;
_hw_output_stencil(0, 0) = _244;
_hw_output_stencil_stream.write(_hw_output_stencil);
```

- Type casting from Boolean to signed integer is different from C.
- Bool in FIRRTL/Verilog is implemented as a 1-bit data type (either wire or reg).
- True (1'b1) → -1 (8'hff), if converted to signed char.

Result mismatch

In the generated FIRRTL:

```
_in_stencil <= _in_stencil_stream_value
node _240 = _in_stencil[0][0]
node _241 = asSInt(_240);
node _242 = lt(_threshold < _241)
node _243 = asSInt(_242) ; _243 is either 0 or -1
node _244 = asSInt(bits(mul(_243, _240), 7, 0))
_hw_output_stencil[0][0] <= _244
_hw_output_stencil_stream_value <= _hw_output_stencil
```

In the converted Verilog-HDL using firrtl utility:

```
assign _240 = $signed(in_stencil_0_0);
assign _GEN_58 = {{24{_240[7]}},_240};
assign _242 = $signed(_threshold) < $signed(_GEN_58);
assign _243 = $signed(_242) ; // _243 is either 0 or -1
assign _GEN_59 = {8{_243}};  // _GEN_59 is either all-zero or all-one
assign _GEN_60 = $signed(_GEN_59) * $signed(in_stencil_0_0);
assign _GEN_61 = _GEN_60[7:0];
assign _244 = $signed(_GEN_61);
assign _hw_output_stencil_0_0 = _244;
always@(posedge clock)
   _hw_output_stencil_stream_value <= _hw_output_stencil_0_0;
```

Issue – Type Casting (Unsigned to Int)

In Halide:

```
hw_output(x, y) = in(x, y) * scale; // in(x, y) is uchar, scale is int.
```

In the generated HLS C:

```
_in_stencil = _in_stencil_stream.read();
uint8_t _240 = _in_stencil(0, 0);
int32_t _241 = (int32_t)(_240); // uchar to int: 128 → 128
int32_t _242 = _241 * _scale;
_hw_output_stencil(0, 0) = _242;
_hw_output_stencil_stream.write(_hw_output_stencil);
```

- Similar issue of converting Boolean to Integer.
- In C: (unsigned char)128 → (int)128
- In FIRRTL/Verilog: 128 (8'h80) → -128 (32'hffffff80) if converted to signed.

Result mismatch

In the generated FIRRTL:

```
_in_stencil <= _in_stencil_stream_value
node _240 = _in_stencil[0][0]
node _241 = asSInt(_240)  ; uchar to int 128 → -128
node _242 = asSInt(bits(mul(_241, _scale), 31, 0))
_hw_output_stencil[0][0] <= _242
_hw_output_stencil_stream_value <= _hw_output_stencil
```

In the converted Verilog-HDL using firrtl utility:

```
assign _240 = $signed(in_stencil_0_0);
assign _GEN_58 = {{24{_240[7]}},_240};
assign _GEN_59 = $signed(_GEN_58) * $signed(_scale);
assign _GEN_60 = _GEN_59[31:0];
assign _242 = $signed(_GEN_60);
assign _hw_output_stencil_0_0 = _242;
always@(posedge clock)
   _hw_output_stencil_stream_value <= _hw_output_stencil_0_0;
```

Issue –Type Casting (Solution)

In Halide:

```
hw_output(x, y) = (in(x, y) > threshold) * in(x, y); // Boolean type is used in math.
```

In the generated HLS C:

```
_in_stencil = _in_stencil_stream.read();
int8_t _240 = _in_stencil(0, 0);
int32_t _241 = (int32_t)(_240);
bool _242 = _threshold < _241;
int8_t _243 = (int8_t) (_242);
int8_t _244 = _243 * _240;
_hw_output_stencil(0, 0) = _244;
_hw_output_stencil_stream.write(_hw_output_stencil);
```

- Perform Bit-width extension before type conversion.
- Use pad() operator with asSInt() and asUInt()
- For wider to narrower casting, simply truncate lower bits and perform type conversion.

Result match

In the generated FIRRTL:

```
_in_stencil <= _in_stencil_stream_value
node _240 = _in_stencil[0][0]
node _241 = asSInt(pad(_240, 32)) ; zero-ext, as _in_stencil is unsigned
node _242 = lt(_threshold < _241)
node _243 = asSInt(pad(_242, 8)) ; zero-ext, as _242 is unsigned
node _244 = asSInt(bits(mul(_243, _240), 7, 0))
_hw_output_stencil[0][0] <= _244
_hw_output_stencil_stream_value <= _hw_output_stencil
```

In the converted Verilog-HDL using firrtl utility:

```
assign _GEN_58 = {{24'd0},in_stencil_0_0};
assign _241 = $signed(_GEN_58);
assign _242 = $signed(_threshold) < $signed(_241);
assign _GEN_59 = {{7'd0},_242};
assign _243 = $signed(_GEN_59);
assign _GEN_60 = $signed(_243) * $signed(in_stencil_0_0);
assign _GEN_61 = _GEN_60[7:0];
assign _244 = $signed(_GEN_61);
assign _hw_output_stencil_0_0 = _244;
always@(posedge clock)
   _hw_output_stencil_stream_value <= _hw_output_stencil_0_0;
```

Issue – Result Bitwidth of Operation

- FIRRTL operation has its own rule for result width:
  - Result width of operation (i.e. Add/Sub/Mul) is defined so that no overflow happens.
  - Ex) unsigned 8-bit + unsigned 8-bit = unsigned 9-bit
- Workaround: Explicitly extract proper bits for the result to be matched with C.
  - Ex) node result = tail(add(a_u8bit, b_u8bit), 1) ; // uchar + uchar = uchar
  - Ex) node result = bits(mul(a_u32bit, b_u32bit), 31, 0) ; // uint * uint = uint

Issue – Result Type of Operation

- FIRRTL operation has its own rule for result type:
  - Result of bitwise operation (ie. And/Or/Xor) is always unsigned.
  - Result of Sub is always signed.
  - If one of operand in Add/Mult is signed, result is signed.
  - bits(), tail() operator results are always unsigned.
- Workaround: Explicitly cast the result back to the original type.

Issue – Dynamic Shifter

- dshl(n, e) and dshr(n, e) should have ‘e’ unsigned.
- Workaround: Force casting ‘e’ to unsigned.

- dshl(n, e) operator has internal limitation that bitwidth of ‘e’ should not be larger that 20 bit.
- Workaround: Force casting ‘e’ to 8-bit unsigned. Shifting from 0 up to 255 bits is practically enough.

- Note: In C, behavior on negative value is undefined.

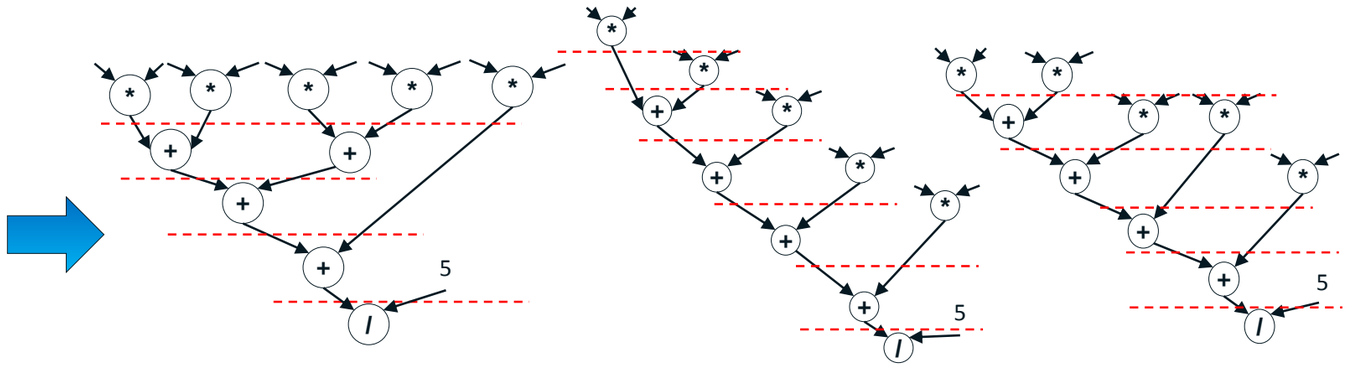
Issue – Multiplexer

- mux(s, t, f) requires both ‘t’ are ‘f’ same type, either signed or unsigned.
- Workaround: Force casting ‘t’ and ‘f’ to result type.
  - Ex) asSInt(mux(s, asSInt(t), asSInt(f)))

Future Work – Loop Pipelining

- Currently Loop body is scheduled in one cycle.
- How to implement Loop Pipelining?
  - Resource mapping? Ex) Register file or memory
  - Resource sharing? Ex) single multiplier II=5
  - Resource duplication? Ex) Use 5 single port memories to make II=1
  - How to support multi-cycle operator? Ex) Divider, Modulus, Multiplier, etc
  - How to provide technology library characteristic?

```
blur_x.stencil(0, 0) = ((((((in.stencil(0, 0)*weight.tap.stencil(0, 0))
                          + (in.stencil(1, 0)*weight.tap.stencil(0, 1)))
                          + (in.stencil(2, 0)*weight.tap.stencil(0, 2)))
                          + (in.stencil(3, 0)*weight.tap.stencil(0, 3)))
                          + (in.stencil(4, 0)*weight.tap.stencil(0, 4)))/(uint8)5)
```

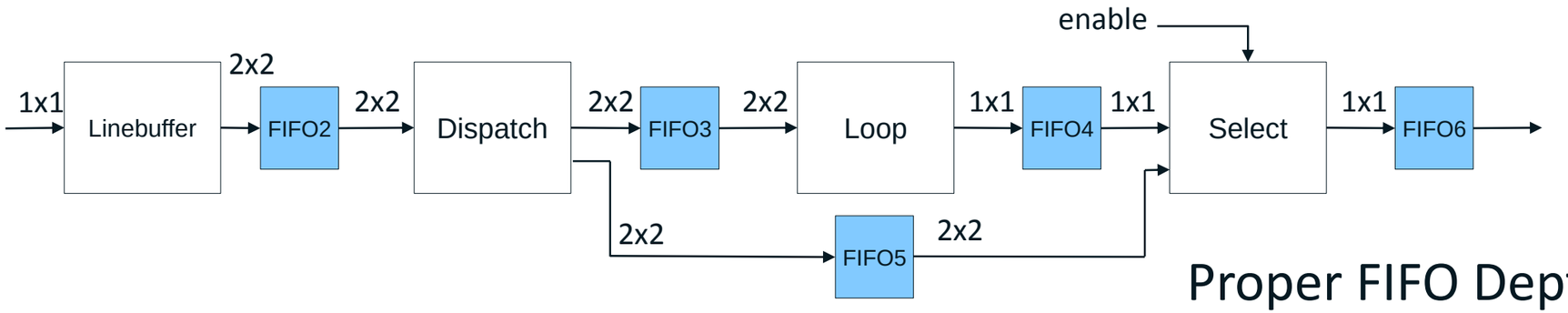


Using register file (II=1)

Using 1-port memory (II=5)

Using 2-port memory (II=3)

Future Work – Automatic FIFO-Depth Calculation



Proper FIFO Depth?

If pipeline depth of Loop is 5:
<5: deadlock
5,6: less than max performance
7: maximum performance
>7: waste

Current solution:

```
in.fifo_depth(hw_output, 7);
```

Specify it manually.