Lab #A
Ch7. Memory Architectures

Team 9

R09922187 鄭又愷
R09922150 洪崗竣
R09922190 王祥任

Outline

- 7.1. Memory-based Shift Register
- 7.2. Memory Organization
- 7.3. Widening the Word Width of Memories
- 7.4. Caching
- Achieving Multi-port memory performance on Single-port memory with coding technique

Outline

- **7.1. Memory-based Shift Register**
- 7.2. Memory Organization
- 7.3. Widening the Word Width of Memories
- 7.4. Caching
- Achieving Multi-port memory performance on Single-port memory with coding technique

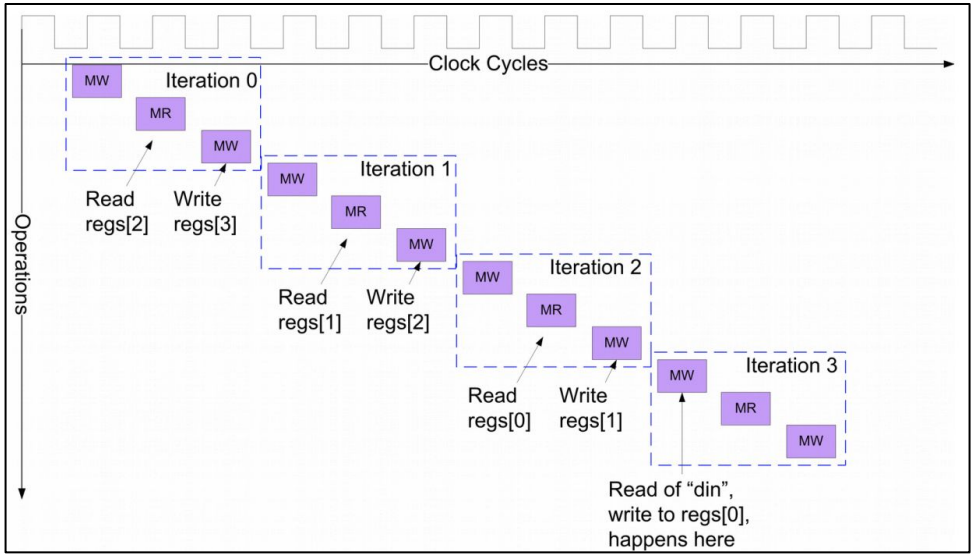
7.1 Memory-based Shift Register

- Classic Shift Register (Naive coding style)
- Circular Shift Register (Optimized one)

Classic Shift Register

```
  static dType regs[N_REGS];
#pragma HLS RESOURCE variable=regs core=RAM_1P_BRAM

SHIFT:
  for (int i = N_REGS - 1; i >= 0; i--) {
    if (i == 0)
      regs[i] = din;
    else
      regs[i] = regs[i - 1];
  }
```

Pragma for using single-port BRAM to synthesis the memory

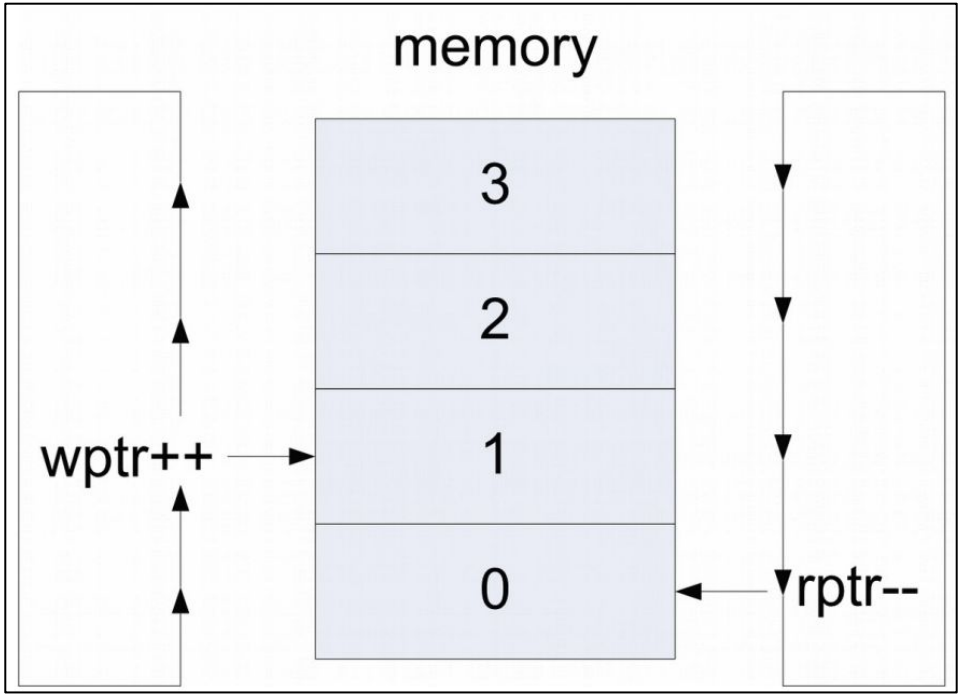


Timeline from Mentor

- When SHIFT occurs, we have to move all the elements in memory, which needs a lot of memory access. (It's OK for register but not for memory)

Circular Shift Register

- We don’t want so much memory access for writing and moving elements.



Circular Shift Register (cont.)

- Instead of moving elements, using pointer to maintain the index of elements.
- Need more time for reading, but for writing, it needs one cycle only.

```
void operator <<(T data){
  mem[wptr] = data;
  wptr++;
  if(wptr==N)
    wptr=0;
}
T operator [](ac_int<ac::log2_ceil<N>::val,false> idx){
  rptr = (wptr-1-idx);
  if(rptr<0)
    rptr = rptr+N;
  return mem[rptr];
}
```

```
void circular_shift_reg(dType din, dType dout[N_REGS]) {

  static circular_shift<dType, N_REGS> regs;

SHIFT:
  regs << din;

WRITE:
  for (int i = 0; i < N_REGS; i++) {
        dout[i] = regs[i];
  }
}
```

Comparison - Latency

```
+ Latency:
    * Summary:
    +---------+---------+----------+----------+-----+-----+---------+
    |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
    |   min   |   max   |    min   |    max   | min | max |   Type  |
    +---------+---------+----------+----------+-----+-----+---------+
    |       22|       30| 0.110 us | 0.150 us |   22|   30|   none  |
    +---------+---------+----------+----------+-----+-----+---------+

    + Detail:
        * Instance:
        N/A

        * Loop:
        +----------+---------+---------+----------+-----------+-----------+-----+----------+
        |          | Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
        | Loop Name|   min   |   max   | Latency  | achieved  |   target  | Count| Pipelined|
        +----------+---------+---------+----------+-----------+-----------+-----+----------+
        |- SHIFT   |        8|       16|  2 ~ 4   |          -|          -|    4|    no    |
        |- WRITE   |       12|       12|        3|          -|          -|    4|    no    |
        +----------+---------+---------+----------+-----------+-----------+-----+----------+
```

```
+ Latency:
    * Summary:
    +---------+---------+----------+----------+-----+-----+---------+
    |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
    |   min   |   max   |    min   |    max   | min | max |   Type  |
    +---------+---------+----------+----------+-----+-----+---------+
    |       22|       22| 0.110 us | 0.110 us |   22|   22|   none  |
    +---------+---------+----------+----------+-----+-----+---------+

    + Detail:
        * Instance:
        N/A

        * Loop:
        +----------+---------+---------+----------+-----------+-----------+-----+----------+
        |          | Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
        | Loop Name|   min   |   max   | Latency  | achieved  |   target  | Count| Pipelined|
        +----------+---------+---------+----------+-----------+-----------+-----+----------+
        |- WRITE   |       20|       20|         5|          -|          -|    4|    no    |
        +----------+---------+---------+----------+-----------+-----------+-----+----------+
```

(L) Classic Shift Register (R) Circular Shift Register

Comparison - Utilization

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	42	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	1	-	0	0	0
Multiplexer	-	-	-	119	-
Register	-	-	88	-	-
Total	1	0	88	161	0
Available	280	220	106400	53200	0
Utilization (%)	~0	0	~0	~0	0

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	155	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	1	-	0	0	0
Multiplexer	-	-	-	65	-
Register	-	-	114	-	-
Total	1	0	114	220	0
Available	280	220	106400	53200	0
Utilization (%)	~0	0	~0	~0	0

(L) Classic Shift Register (R) Circular Shift Register

Outline

- 7.1. Memory-based Shift Register
- **7.2. Memory Organization**
- 7.3. Widening the Word Width of Memories
- 7.4. Caching
- Achieving Multi-port memory performance on Single-port memory with coding technique

7.2 Memory Organization

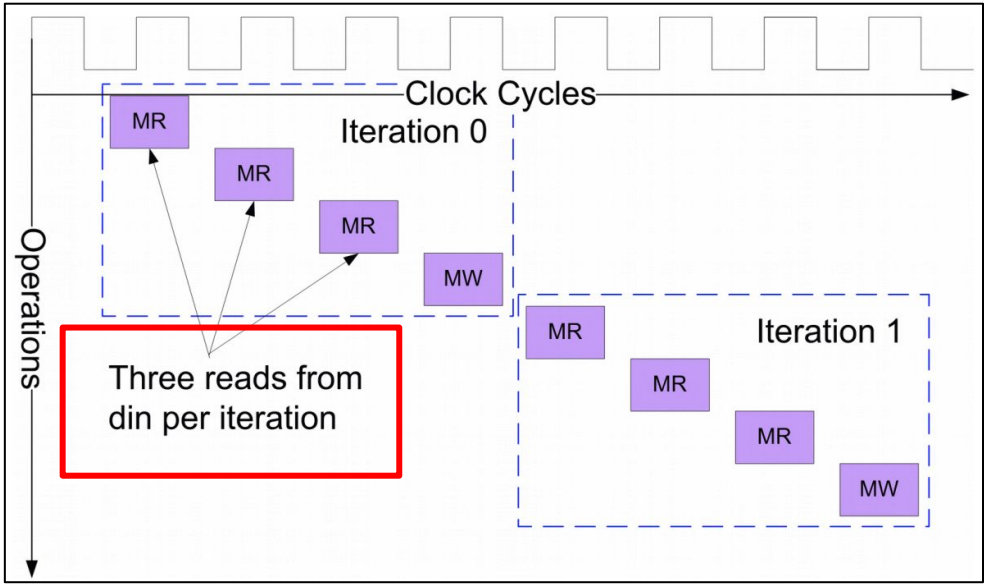
- Automatic Interleave by Xilinx HLS Pragma
- Manual Interleave (Random Access)
- Manual Interleave (Sequential Access)

```
#include "interleave.h"
void interleave(ac_int<8> x_in[NUM_WORDS], ac_int<8> y[NUM_WORDS/3],
                bool load){
  static ac_int<8> x[NUM_WORDS];
  int idx = 0;

  if(load)
    for(int i=0;i<NUM_WORDS;i+=1)
      x[i] = x_in[i];
  else
    for(int i=0;i<NUM_WORDS;i+=3)
      y[idx++] = x[i]+x[i+1]+x[i+2];

}
```

```
void interleave(
        ap_int<8> x_in[NUM_WORDS],
        ap_int<8> y[NUM_WORDS / 3],
        bool load) {

  static ap_int<8> x[NUM_WORDS];

  int idx = 0;

  if (load)
      // Load all the value of x_in into x
    for (int i = 0; i < NUM_WORDS; i += 1)
#pragma HLS PIPELINE II=1
        x[i] = x_in[i];
  else
      // Return the sum of 3 continuous elements in x;
      // y[0] = x[0] + x[1] + x[2];
      // y[1] = x[3] + x[4] + x[5];
      // y[2] = x[6] + x[7] + x[8];
    for (int i = 0; i < NUM_WORDS; i += 3)
#pragma HLS PIPELINE II=1
        y[idx++] = x[i] + x[i + 1] + x[i + 2];
}
```

Access Pattern of Top Function

```
if(load)
  for(int i=0;i<NUM_WORDS;i+=1)
    x[i] = x_in[i];
else
  for(int i=0;i<NUM_WORDS;i+=3)
    y[idx++] = x[i]+x[i+1]+x[i+2];
```



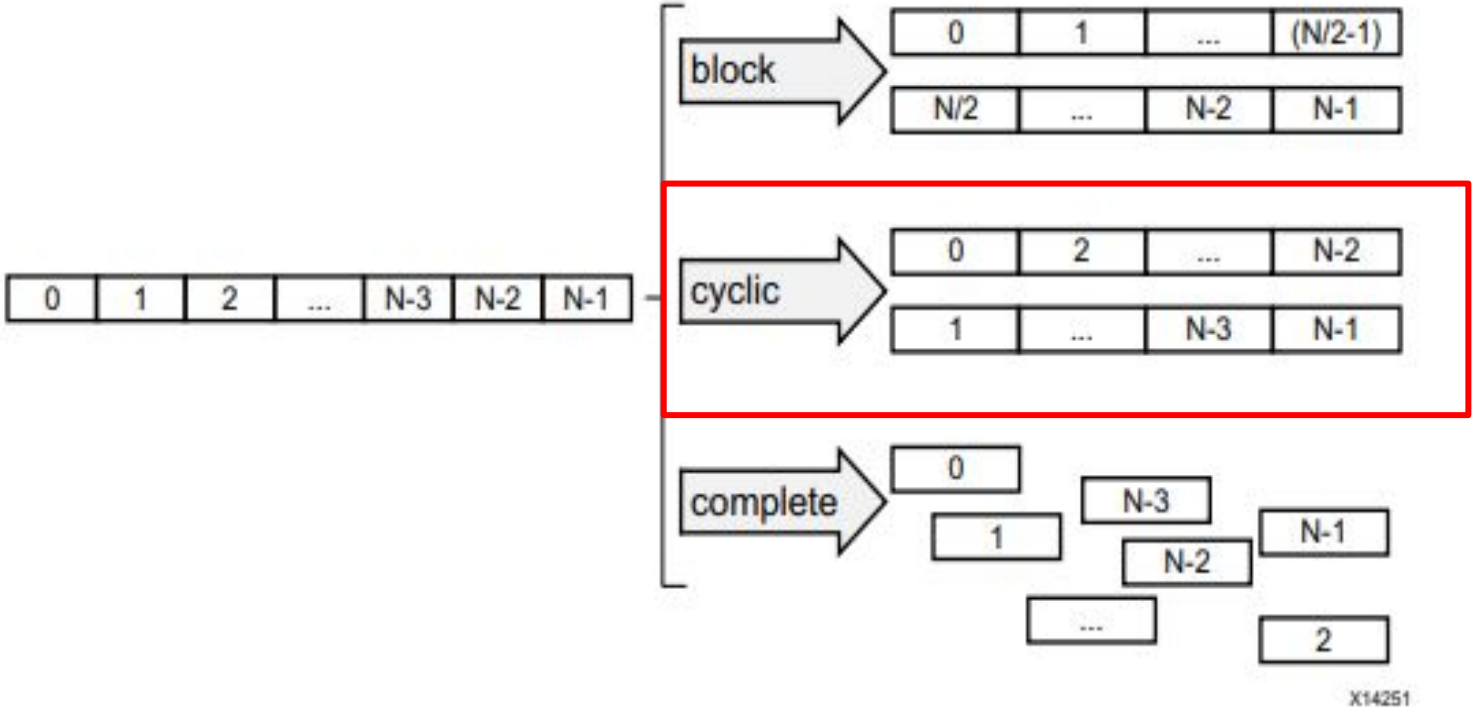
Pattern of Write Out

Automatic Interleave

- ARRAY_PARTITION
- Cyclic, Factor = 3

```
void interleave(
                ap_int<8> x_in[NUM_WORDS],
                ap_int<8> y[NUM_WORDS / 3],
                bool load) {
#pragma HLS RESOURCE variable=x_in core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=y core=RAM_1P_BRAM

  static ap_int<8> x[NUM_WORDS];
#pragma HLS RESOURCE variable=x core=RAM_1P_BRAM
#pragma HLS ARRAY_PARTITION variable=x cyclic factor=3 dim=1

  int idx = 0;

  if (load)
LOAD:
    for (int i = 0; i < NUM_WORDS; i += 1)
#pragma HLS PIPELINE II=1
        x[i] = x_in[i];
  else
WRITE:
    for (int i = 0; i < NUM_WORDS; i += 3)
#pragma HLS PIPELINE II=1
        y[idx++] = x[i] + x[i + 1] + x[i + 2];
}
```

Automatic Interleave (cont.)



Manual Interleave (Random Access)

```
void interleave_manual_rnd(ap_int<8> x_in[NUM_WORDS],
                           ap_int<8> y[NUM_WORDS / 3], bool load) {
#pragma HLS RESOURCE variable=x_in core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=y core=RAM_1P_BRAM

  static interleave_mem_rnd<ap_int<8>, NUM_WORDS> x;
  int idx = 0;

  if (load)
LOAD:
    for (int i = 0; i < NUM_WORDS; i += 1)
#pragma HLS PIPELINE II=1
        x.write_rnd(i, x_in);
  else
WRITE:
    for (int i = 0; i < NUM_WORDS; i += 3)
#pragma HLS PIPELINE II=1
      y[idx++] = x.read_rnd(i, 0) + x.read_rnd(i, 1) + x.read_rnd(i, 2);
}
```

```
class interleave_mem_rnd {
 public:
  T x0[N / 3];
  T x1[N / 3];
  T x2[N / 3];

 public:
  interleave_mem_rnd() {
#pragma HLS RESOURCE variable=x0 core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=x1 core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=x2 core=RAM_1P_BRAM
  }

  void write_rnd(ap_uint<ADDRESS_BITWIDTH> i, T x_in[N]);
  T read_rnd(ap_uint<ADDRESS_BITWIDTH> i, int offset);
};
```

Manual Interleave (Random Access) (cont.)

```
template <typename T, int N>
void interleave_mem_rnd<T, N>::write_rnd(ap_uint<ADDRESS_BITWIDTH> i,
                                         T x_in[N]) {
  T tmp = x_in[i];
  switch (i % 3) {
    case 0:
      x0[i / 3] = tmp;
      break;
    case 1:
      x1[i / 3] = tmp;
      break;
    case 2:
      x2[i / 3] = tmp;
      break;
  }
}
```

```
template <typename T, int N>
T interleave_mem_rnd<T, N>::read_rnd(ap_uint<ADDRESS_BITWIDTH> i,
                                     int offset) {

        // Force function being merged into the FSM of WRITE
#pragma HLS INLINE

  T tmp = 0;
  switch (offset) {
    case 0:
      tmp = x0[i / 3];
      break;
    case 1:
      tmp = x1[i / 3];
      break;
    case 2:
      tmp = x2[i / 3];
      break;
  }
  return tmp;
}
```

Or read_rnd will be synthesized as an individual module, which will increase the interval.

Manual Interleave (Sequential Access)

```
void interleave_manual_seq(ap_int<8> x_in[NUM_WORDS],
                           ap_int<8> y[NUM_WORDS / 3], bool load) {
#pragma HLS RESOURCE variable=x_in core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=y core=RAM_1P_BRAM

  static interleave_mem_seq<ap_int<8>, NUM_WORDS> x;
  int idx = 0;

  if (load)
LOAD:
    for (int i = 0; i < NUM_WORDS; i += 1)
#pragma HLS PIPELINE II=1
        x.write_seq(i, x_in);
  else
WRITE:
    for (int i = 0; i < NUM_WORDS / 3; i += 1)
#pragma HLS PIPELINE II=1
        // Some Modification Here
        // Follow the description in book but not code
        y[idx++] = x.read_seq(i, 0) + x.read_seq(i, 1) + x.read_seq(i, 2);
}
```

```
class interleave_mem_seq {
 public:
  T x0[N / 3];
  T x1[N / 3];
  T x2[N / 3];
  ap_uint<ADDRESS_BITWIDTH> idx;
  ap_uint<2> sel;

 public:
  interleave_mem_seq() {
#pragma HLS RESOURCE variable = x0 core = RAM_1P_BRAM
#pragma HLS RESOURCE variable = x1 core = RAM_1P_BRAM
#pragma HLS RESOURCE variable = x2 core = RAM_1P_BRAM

        idx = 0;
    sel = 0;
  }

  void write_seq(ap_uint<ADDRESS_BITWIDTH> i, T x_in[N]);
  T read_seq(ap_uint<ADDRESS_BITWIDTH> i, int offset);
};
```

Manual Interleave (Sequential Access)

WRITE

```
template <typename T, int N>
void interleave_mem_seq<T, N>::write_seq(ap_uint<ADDRESS_BITWIDTH> i,
                                         T x_in[N]) {
  int tmp = x_in[i];
  switch (sel++) {
    case 0:
      x0[idx] = tmp;
      break;
    case 1:
      x1[idx] = tmp;
      break;
    case 2:
      x2[idx++] = tmp;
      break;
  }

  if (idx == N / 3) idx = 0;
  if (sel == 3) sel = 0;
}
```

Add (sel++) VS Modulo (i%3)

No Operation VS Divide (i / 3)

Add (idx++) VS Divide (i / 3)

READ

```
template <typename T, int N>
T interleave_mem_seq<T, N>::read_seq(ap_uint<ADDRESS_BITWIDTH> i,
                                     int offset) {
  T tmp = 0;
  switch (offset) {
    case 0:
      tmp = x0[i];
      break;
    case 1:
      tmp = x1[i];
      break;
    case 2:
      tmp = x2[i];
      break;
  }

  return tmp;
}
```

No “HLS INLINE” here since it will be inlined automatically.

No division here

Comparison - Timing

```
+ Timing:
    * Summary:
    +--------+---------+---------+-----------+
    | Clock  | Target  | Estimated | Uncertainty|
    +--------+---------+---------+-----------+
    |ap_clk  | 5.00 ns | 3.064 ns |  0.62 ns  |
    +--------+---------+---------+-----------+
```

Auto By Xilinx HLS Pragma

```
+ Timing:
    * Summary:
    +--------+---------+---------+-----------+
    | Clock  | Target  | Estimated | Uncertainty|
    +--------+---------+---------+-----------+
    |ap_clk  | 5.00 ns | 3.064 ns |  0.62 ns  |
    +--------+---------+---------+-----------+
```

Manual (Random Access)

```
+ Timing:
    * Summary:
    +--------+---------+---------+-----------+
    | Clock  | Target  | Estimated | Uncertainty|
    +--------+---------+---------+-----------+
    |ap_clk  | 5.00 ns | 2.664 ns |  0.62 ns  |
    +--------+---------+---------+-----------+
```

Manual (Sequential Access)

Comparison - Latency

```
Latency:
  * Summary:
  +---------+---------+-----------+-----------+-----+-----+---------+
  |  Latency (cycles) |   Latency (absolute)  |  Interval | Pipeline|
  |   min   |   max   |    min    |    max    | min | max |   Type  |
  +---------+---------+-----------+-----------+-----+-----+---------+
  |       7|       12| 35.000 ns | 60.000 ns |    7|   12|   none  |
  +---------+---------+-----------+-----------+-----+-----+---------+

  + Detail:
      * Instance:
      N/A

      * Loop:
      +----------+---------+---------+----------+-----------+-----------+----------+
      |          | Latency (cycles) | Iteration| Initiation Interval | Trip |          |
      | Loop Name|   min   |   max   | Latency  | achieved  |   target  | Count| Pipelined|
      +----------+---------+---------+----------+-----------+-----------+----------+
      |- LOAD    |      10|      10|         3|          1|          1|     9|    yes   |
      |- WRITE   |       5|       5|         4|          1|          1|     3|    yes   |
      +----------+---------+---------+----------+-----------+-----------+----------+
```

Without any optimization,
latency is between 12 ~ 14 cycles.

Comparison - Utilization

	BRAM	FF	LUT
Auto	3	211	329
Manual (Random Access)	3	237	405
Manual (Sequential Access)	3	260	387

```
* Memory:
+---------+------------------+---------+---+----+-----+------+-----+------+-------------+
|  Memory |      Module      | BRAM_18K| FF| LUT| URAM| Words| Bits| Banks| W*Bits*Banks|
+---------+------------------+---------+---+----+-----+------+-----+------+-------------+
|x_V_0_U  |interleave_x_V_0  |        1|  0|   0|    0|     3|    8|     1|           24|
|x_V_1_U  |interleave_x_V_0  |        1|  0|   0|    0|     3|    8|     1|           24|
|x_V_2_U  |interleave_x_V_0  |        1|  0|   0|    0|     3|    8|     1|           24|
+---------+------------------+---------+---+----+-----+------+-----+------+-------------+
|Total    |                  |        3|  0|   0|    0|     9|   24|     3|           72|
+---------+------------------+---------+---+----+-----+------+-----+------+-------------+
```

Outline

- 7.1. Memory-based Shift Register
- 7.2. Memory Organization
- **7.3. Widening the Word Width of Memories**
- 7.4. Caching
- Achieving Multi-port memory performance on Single-port memory with coding technique

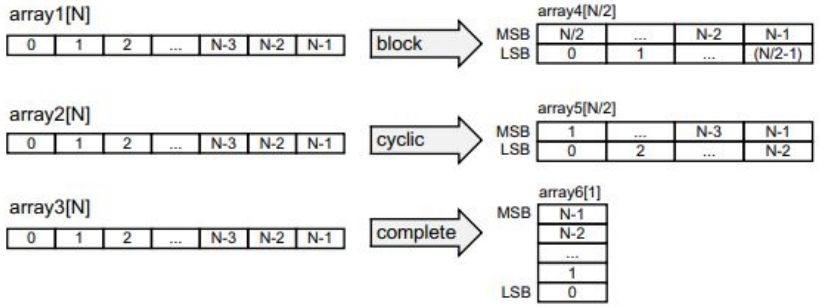
7.3 Widening the Word Width of Memories

- Xilinx HLS Pragma
- Manual (Sequential Access)

```
void interleave(
        ap_int<8> x_in[NUM_WORDS],
        ap_int<8> y[NUM_WORDS / 3],
        bool load) {

  static ap_int<8> x[NUM_WORDS];

  int idx = 0;

  if (load)
      // Load all the value of x_in into x
    for (int i = 0; i < NUM_WORDS; i += 1)
#pragma HLS PIPELINE II=1
        x[i] = x_in[i];
  else
      // Return the sum of 3 continuous elements in x;
      // y[0] = x[0] + x[1] + x[2];
      // y[1] = x[3] + x[4] + x[5];
      // y[2] = x[6] + x[7] + x[8];
    for (int i = 0; i < NUM_WORDS; i += 3)
#pragma HLS PIPELINE II=1
        y[idx++] = x[i] + x[i + 1] + x[i + 2];
}
```

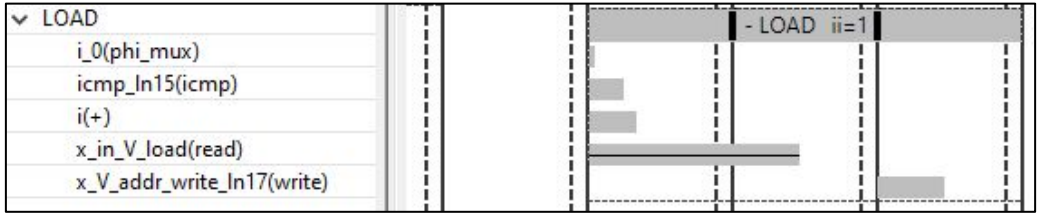
Xilinx HLS Pragma

- ARRAY_RESHAPE
- Cyclic, factor = 3
- For WRITE, II is 1.
- For LOAD, II is 3.




```
void word_width(ap_int<8> x_in[NUM_WORDS], ap_int<8> y[NUM_WORDS / 3],
                        bool load) {
#pragma HLS RESOURCE variable=x_in core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=y core=RAM_1P_BRAM

  static ap_int<8> x[NUM_WORDS];
#pragma HLS ARRAY_RESHAPE variable=x cyclic factor=3 dim=1
#pragma HLS RESOURCE variable=x core=RAM_1P_BRAM

  int idx = 0;

  if (load)
LOAD:
        for (int i = 0; i < NUM_WORDS; i += 1)
#pragma HLS PIPELINE II=1
                x[i] = x_in[i];
  else
WRITE:
        for (int i = 0; i < NUM_WORDS / 3; i += 1)
#pragma HLS PIPELINE II=1
                y[idx++] = x[i*3+0] + x[i*3+1] + x[i*3+2];
}
```

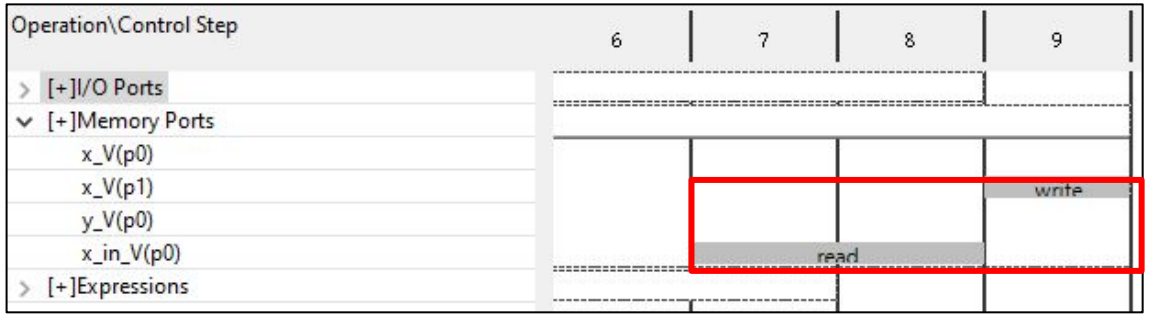
Xilinx HLS Pragma



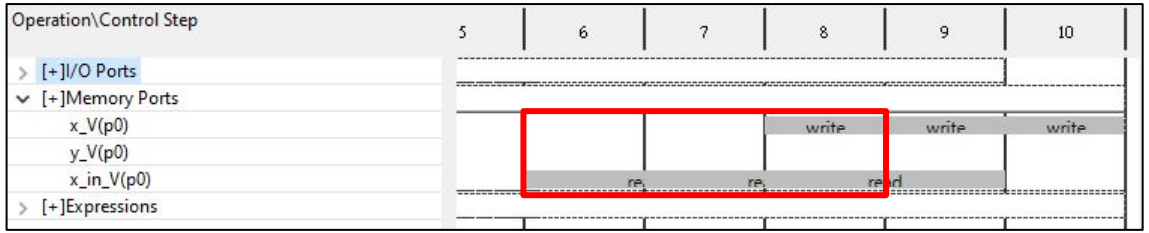
(Without Reshape)
LOAD : II = 1 (Trip Count=9)
WRITE: II = 3

(With Auto Reshape)
LOAD : II = 3 (Trip Count=3)
WRITE: II = 1

Xilinx HLS Pragma



(Without Reshape)
LOAD : II = 1 (Trip Count=9)
WRITE: II = 3



(With Auto Reshape)
LOAD : II = 3 (Trip Count=3)
WRITE: II = 1

Manual (Sequential Access)

```
void word_width_manual(ap_int<8> x_in[NUM_WORDS], ap_int<8> y[NUM_WORDS / 3],
                       bool load) {
#pragma HLS RESOURCE variable=x_in core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=y core=RAM_1P_BRAM

  static word_width_mem<8, NUM_WORDS> x;

  int idx = 0;

  if (load)
LOAD:
    for (int i = 0; i < NUM_WORDS; i += 1)
#pragma HLS PIPELINE II=1
        x.write(i, x_in);
  else
WRITE:
    for (int i = 0; i < NUM_WORDS / 3; i += 1){
#pragma HLS PIPELINE II=1
        y[idx++] = x.read(i, 0) + x.read(i, 1) + x.read(i, 2);
    }
}
```

```
template<int W, int N>
class word_width_mem {
  private:
  ap_uint<W*3> x[N/3];
  ap_uint<ADDRESS_BITWIDTH> idx;
  ap_uint<2> sel_rd;
  ap_uint<2> sel_wr;
  ap_uint<W*3> write3;
  ap_uint<W*3> read3;

  public:
  word_width_mem():sel_rd(0), sel_wr(0){
#pragma HLS RESOURCE variable=x core=RAM_1P_BRAM
  }
  void write(ap_uint<ADDRESS_BITWIDTH> i, ap_int<W> x_in[N]);
  ap_int<W> read(ap_uint<ADDRESS_BITWIDTH> i, const int offset);
};
#include "ww_read_mem.hpp"
#include "ww_write_mem.hpp"

#endif
```

Manual (Sequential Access) - Write

```
template <int W, int N>
void word_width_mem<W, N>::write(ap_uint<ADDRESS_BITWIDTH> i,
                                 ap_int<W> x_in[N]) {
  write3(sel_wr * W + W - 1, sel_wr * W) = x_in[i](W-1, 0);

  sel_wr++;

  if (sel_wr == 3) {
    x[idx++] = write3;
    sel_wr = 0;
  }

  if (idx == N / 3) {
    idx = 0;
  }
}
```

```
template <int W, int N>
ap_int<W> word_width_mem<W, N>::read(ap_uint<ADDRESS_BITWIDTH> i,
                                     const int offset) {

        // Force the call of function being merged
        // into the FSM of WRITE.
        // Critical Path will be too long,
        // but II = 1 is possible
#pragma HLS INLINE


        ap_int<W> tmp = 0;

  if (sel_rd++ == 0)  // read once every 3 calls
    read3 = x[i];

  if (sel_rd == 3) sel_rd = 0;

  switch (offset) {
    case 0:
      tmp = read3(W - 1, 0);
      break;
    case 1:
      tmp = read3(W * 2 - 1, W);
      break;
    case 2:
      tmp = read3(W * 3 - 1, W * 2);
      break;
  }

  return tmp;
}
```

Comparison - Timing

Clock	Target	Estimated	Uncertainty
ap_clk	5.00 ns	3.669 ns	0.62 ns

Auto by Xilinx HLS Pragma

Clock	Target	Estimated	Uncertainty
ap_clk	5.00 ns	10.551 ns	0.62 ns

Manual - Inline

Clock	Target	Estimated	Uncertainty
ap_clk	5.00 ns	4.293 ns	0.62 ns

Manual - No Inline

Comparison - Latency (Auto by Xilinx HLS Pragma)

```
* Summary:
+---------+---------+-----------+-----------+-----+-----+---------+
|  Latency (cycles) |   Latency (absolute)  |  Interval | Pipeline|
|   min   |   max   |    min    |    max    | min | max |   Type  |
+---------+---------+-----------+-----------+-----+-----+---------+
|       7|       12| 35.000 ns | 60.000 ns |    7|   12|   none  |
+---------+---------+-----------+-----------+-----+-----+---------+

+ Detail:
    * Instance:
    N/A

    * Loop:
    +----------+---------+---------+----------+-----------+-----------+------+----------+
    |          |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
    | Loop Name|   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
    +----------+---------+---------+----------+-----------+-----------+------+----------+
    |- WRITE   |       5|       5|         4|          1|          1|     3|    yes   |
    |- LOAD    |      10|      10|         5|          3|          1|     3|    yes   |
    +----------+---------+---------+----------+-----------+-----------+------+----------+
```

Comparison - Latency (Manual - inline)

```
* Summary:
+---------+---------+-----------+----------+-----+-----+---------+
|  Latency (cycles) |  Latency (absolute)  |  Interval | Pipeline|
|   min   |   max   |    min    |    max   | min | max |   Type  |
+---------+---------+-----------+----------+-----+-----+---------+
|        8|       14| 84.408 ns | 0.148 us |    8|   14|   none  |
+---------+---------+-----------+----------+-----+-----+---------+

+ Detail:
    * Instance:
    N/A

    * Loop:
    +----------+---------+---------+----------+-----------+-----------+------+----------+
    |          | Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
    | Loop Name|   min   |   max   | Latency  | achieved  |   target  | Count| Pipelined|
    +----------+---------+---------+----------+-----------+-----------+------+----------+
    |- LOAD    |       11|       11|         4|          1|          1|     9|    yes   |
    |- WRITE   |        5|        5|         4|          1|          1|     3|    yes   |
    +----------+---------+---------+----------+-----------+-----------+------+----------+
```

Comparison - Latency (Manual - No Inline)

```
+ Latency:
    * Summary:
    +---------+---------+-----------+----------+-----+-----+---------+
    |  Latency (cycles) |  Latency (absolute)  |  Interval | Pipeline|
    |   min   |   max   |    min    |    max   | min | max |   Type  |
    +---------+---------+-----------+----------+-----+-----+---------+
    |       13|       31| 65.000 ns | 0.155 us |   13|   31|   none  |
    +---------+---------+-----------+----------+-----+-----+---------+

    + Detail:
        * Instance:
        +-------------------+--------+---------+---------+-----------+-----------+-----+-----+----------+
        |                   |        |  Latency (cycles) |  Latency (absolute)  |  Interval | Pipeline |
        |      Instance     | Module |   min   |   max   |    min    |    max    | min | max |   Type   |
        +-------------------+--------+---------+---------+-----------+-----------+-----+-----+----------+
        |grp_read_r_fu_199  |read_r  |        2|        2| 10.000 ns | 10.000 ns |    1|    1| function |
        +-------------------+--------+---------+---------+-----------+-----------+-----+-----+----------+

        * Loop:
        +----------+---------+---------+----------+-----------+-----------+------+----------+
        |          |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
        | Loop Name|   min   |   max   | Latency  | achieved  |   target  | Count| Pipelined|
        +----------+---------+---------+----------+-----------+-----------+------+----------+
        |- LOAD    |       11|       11|         4|          1|          1|     9|    yes   |
        |- WRITE   |       29|       29|        12|          9|          1|     3|    yes   |
        +----------+---------+---------+----------+-----------+-----------+------+----------+
```

Comparison - Utilization

	BRAM	FF	LUT
Auto	1	211	381
Manual - inline	1	583	1098
Manual - no inline	1	393	909

```
* Memory:
+---------+----------------------+---------+---+----+-----+-----+-----+------+-------------+
|  Memory |        Module        | BRAM_18K| FF| LUT| URAM| Words| Bits | Banks| W*Bits*Banks|
+---------+----------------------+---------+---+----+-----+-----+-----+------+-------------+
|x_x_V_U  |word_width_manualbkb  |        1|  0|   0|    0|    3|   24|     1|           72|
+---------+----------------------+---------+---+----+-----+-----+-----+------+-------------+
|Total    |                      |        1|  0|   0|    0|    3|   24|     1|           72|
+---------+----------------------+---------+---+----+-----+-----+-----+------+-------------+
```

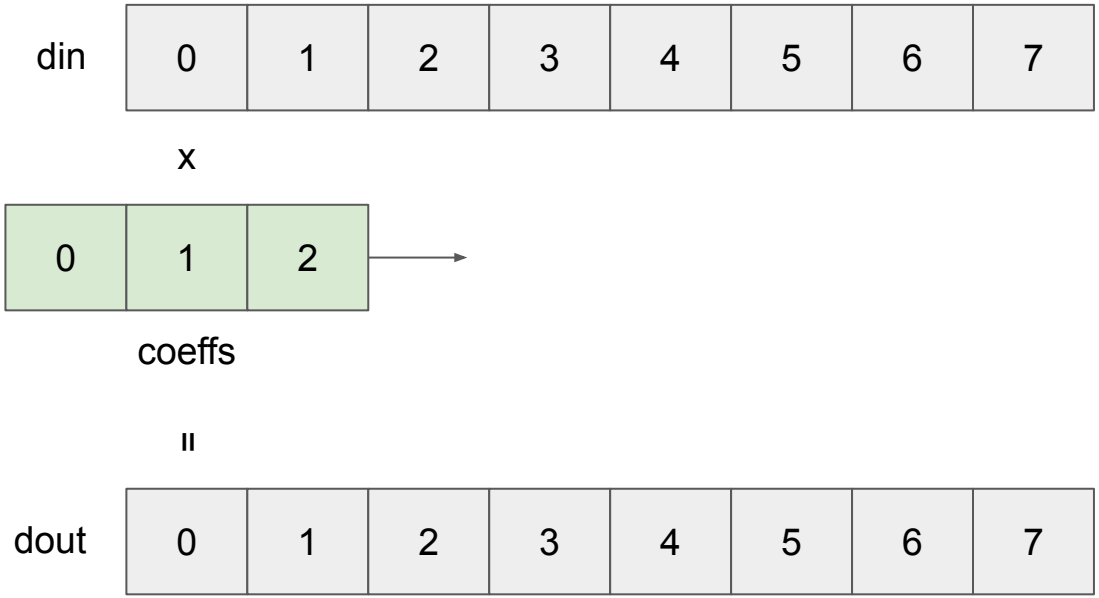
Outline

- 7.1. Memory-based Shift Register
- 7.2. Memory Organization
- 7.3. Widening the Word Width of Memories
- **7.4. Caching**
- Achieving Multi-port memory performance on Single-port memory with coding technique

7.4 Caching

- Windowing of “1D” data stream
  - shift register
- Windowing of “2D” data stream
  - Single-port RAM

Windowing of “1D” data stream - Access Pattern



Windowing of “1D” data stream - Problem

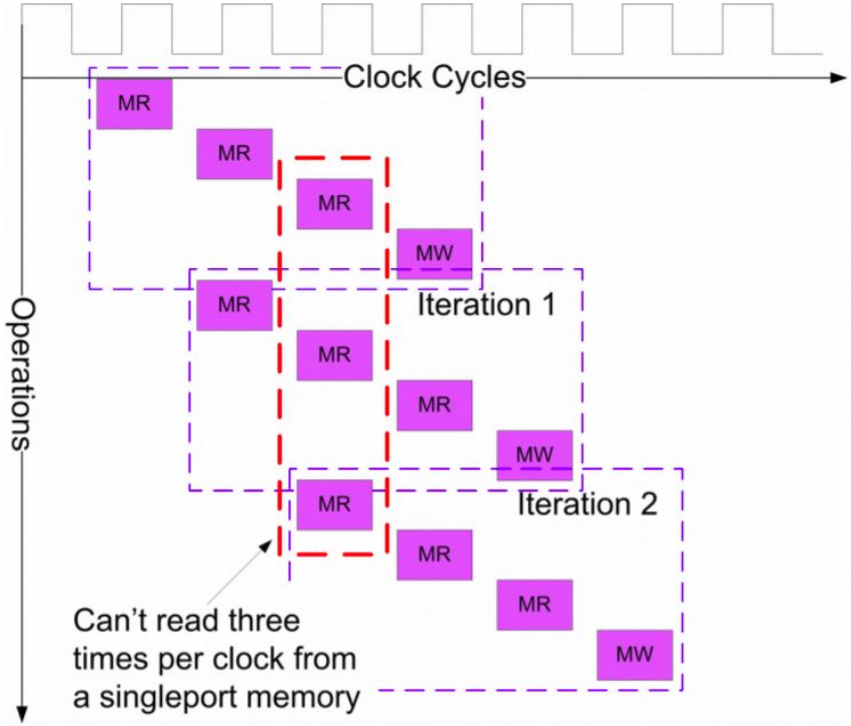
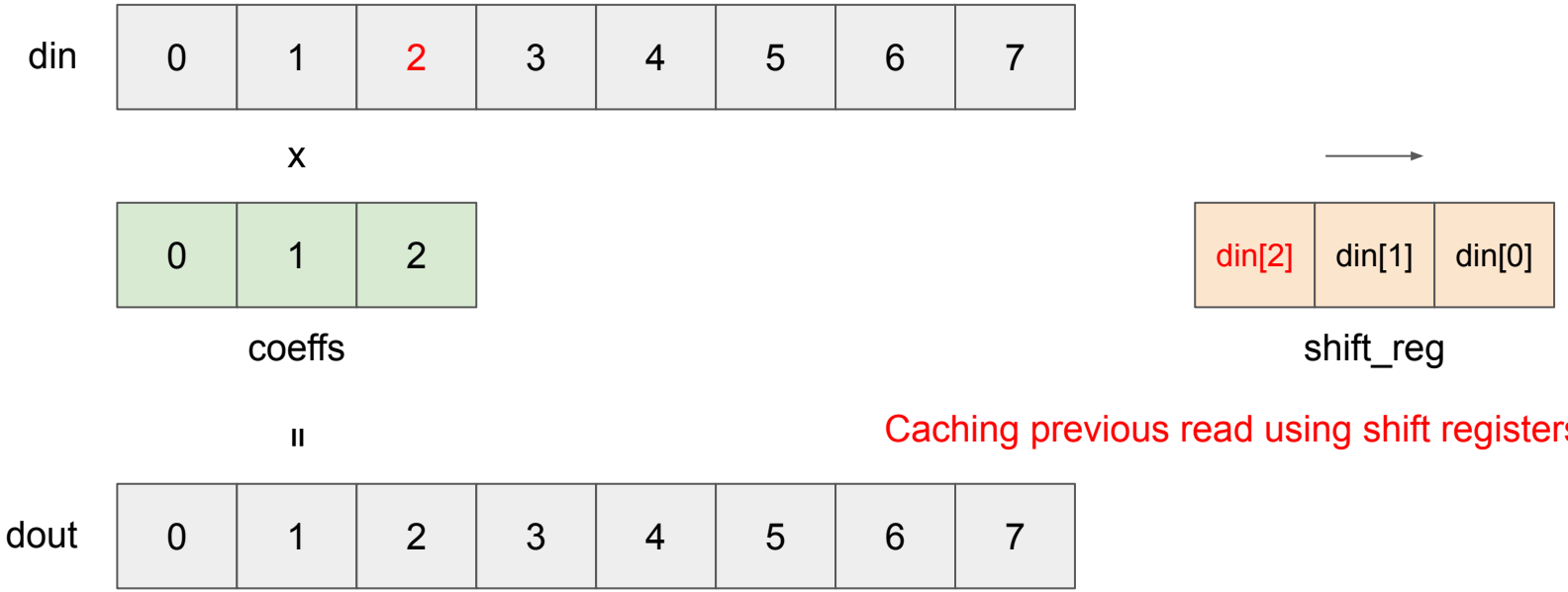


Illustration 117: Failed Schedule for Moving Average with II=1

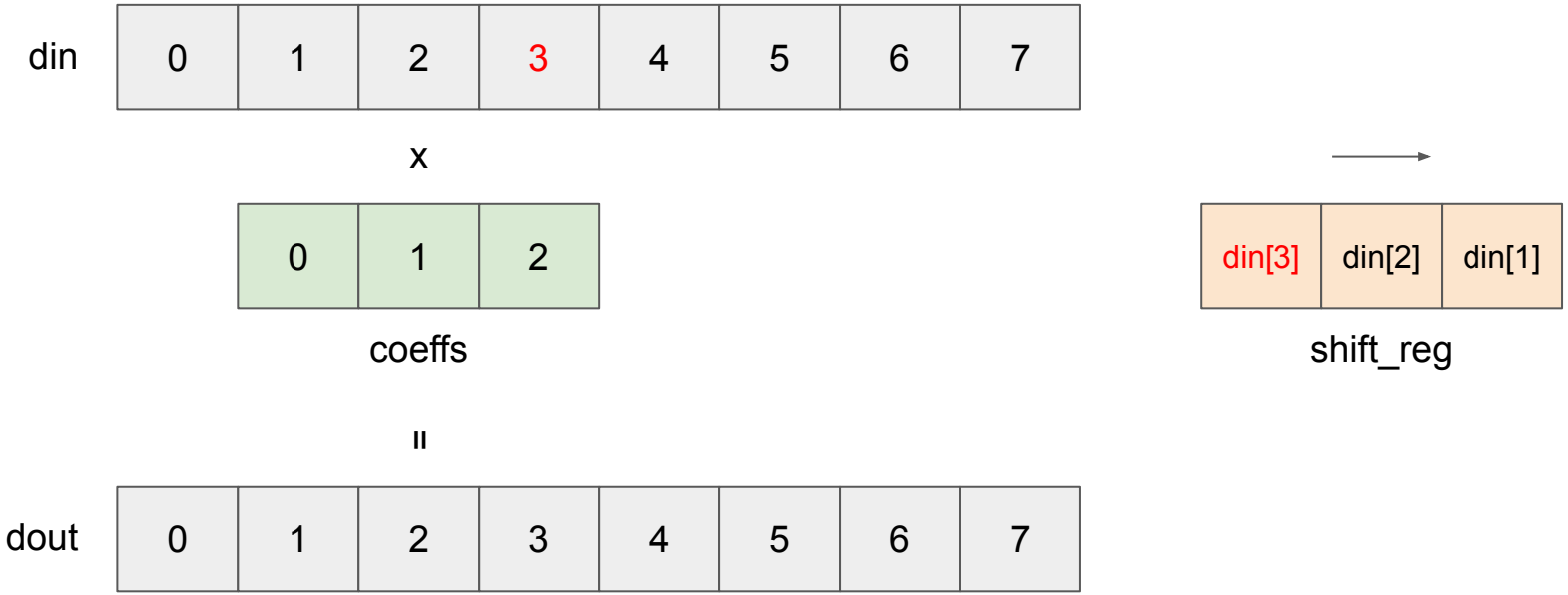
- Read three times per iteration
- Memory only has single port

Windowing of “1D” data stream - Shift register



Caching previous read using shift registers

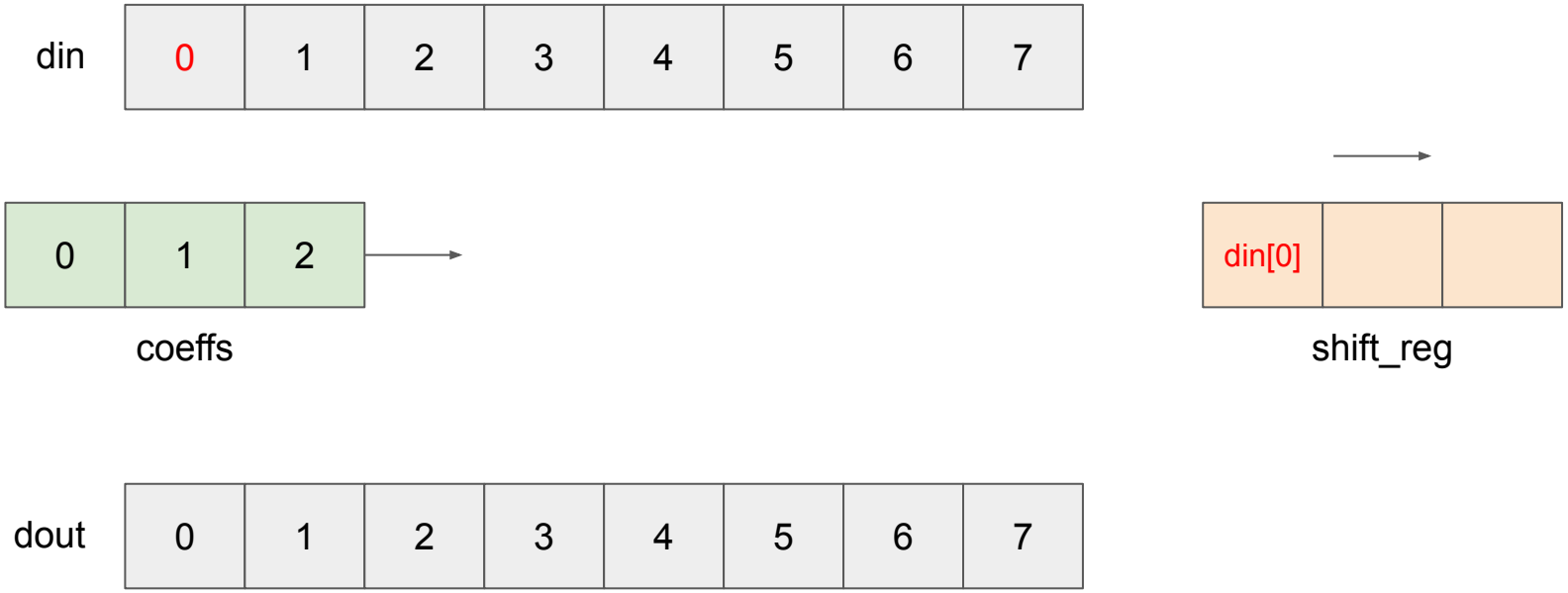
Windowing of “1D” data stream - Shift register (cont.)



Windowing of “1D” data stream - Boundary Conditions

Preparing for dout[0]
Additional iteration in for loop

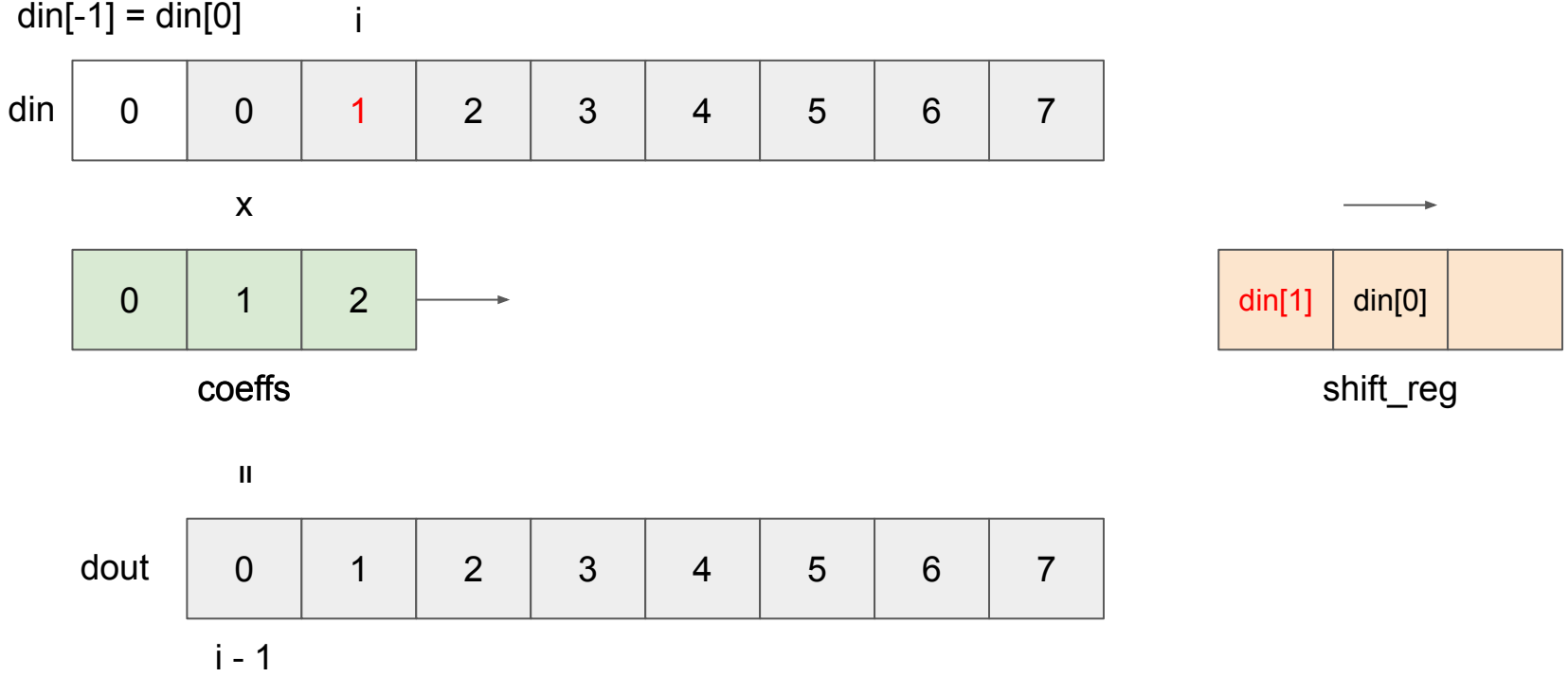
i = 0



Windowing of “1D” data stream - Boundary Conditions

Boundary clipping
din[-1] = din[0]

i = 1



Windowing of “1D” data stream - Clipping function

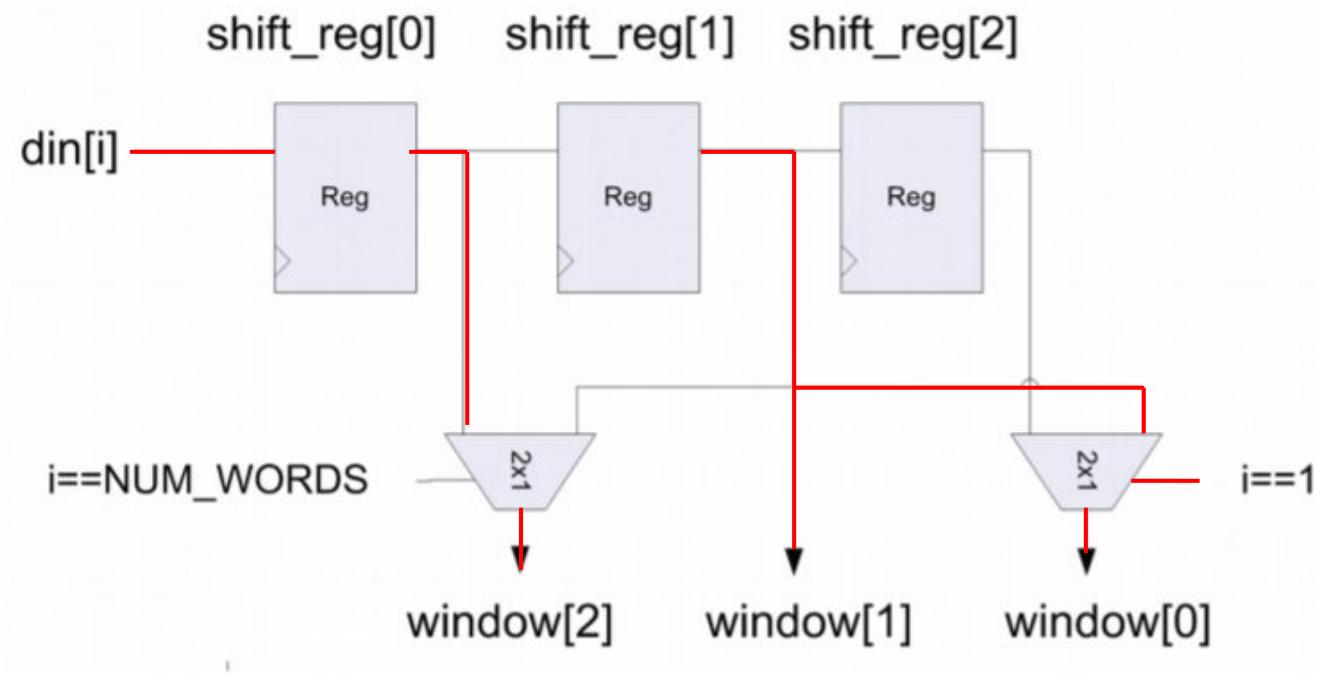
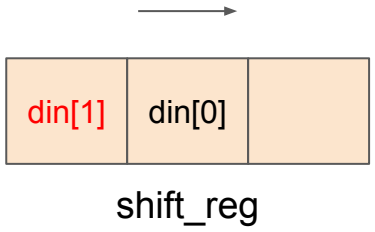


Illustration 119: Clipping Function for 1-D Sliding Window



Windowing of “1D” data stream - Poor Architecture

```
#include "window_1d.h"

int clip(int i) {
    int tmp = i;
    if (tmp < 0)
        tmp = 0;
    else if (tmp > NUM_WORDS - 1)
        tmp = NUM_WORDS - 1;
    return tmp;
}

// can compute a new valud of "dout[i]" every clock cycle
// but the coding style limits the design performance due to a memory bottleneck
void avg(ap_uint<8> din[NUM_WORDS], ap_uint<8> dout[NUM_WORDS]) {
#pragma HLS RESOURCE variable=din core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=dout core=RAM_1P_BRAM

    const ap_ufixed<3, 1> coeffs[3] = {0.25, 0.5, 0.25};
    ap_ufixed<13, 11> tmp;

COMP:
    for (int i = 0; i != NUM_WORDS; i++) {
#pragma HLS PIPELINE II=1
        tmp = din[clip(i - 1)] * coeffs[0] + din[i] * coeffs[1] + din[clip(i + 1)] * coeffs[2];
        dout[i] = tmp.to_int();
    }
}
```

Read1

Read2

Read3

Windowing of “1D” data stream - Windowing

```
// take advantage of array access patterns in order to reduce
// the memory access of "din" to once per clock cycle
void window_avg(ap_uint<8> din[NUM_WORDS],
                ap_uint<8> dout[NUM_WORDS]) {
    const ap_ufixed<3, 1> coeffs[3] = {0.25, 0.5, 0.25};
    // shift registers with three taps, store previous values of "din"
    shift_class<ap_uint<8>, 3> shift_reg;
    // apply the boundary conditions to the sliding shift register
    ap_uint<8> window[3];
    ap_ufixed<13, 11> mac;
    ap_uint<8> din_tmp;

COMP:
    // NUM_WORDS + 1 : "dout[0]" requires additional two reads from "din"
    for (int i = 0; i != NUM_WORDS + 1; i++) {
#pragma HLS PIPELINE II=1
        if (i < NUM_WORDS)  //prevent overread of din
            din_tmp = din[i];
        shift_reg << din_tmp;
        clip_window(shift_reg, i, window);
        mac = window[0] * coeffs[0] + window[1] * coeffs[1] + window[2] * coeffs[2];
        if (i >= 1)  //startup
            dout[i - 1] = mac.to_int();
    }
}
```

```
void clip_window(shift_class<ap_uint<8>, 3> shift_reg,
                 int i, ap_uint<8> window[3]) {
    window[0] = (i == 1) ? shift_reg[1] : shift_reg[2];
    window[1] = shift_reg[1];
    window[2] = (i == NUM_WORDS) ? shift_reg[1] : shift_reg[0];
}
```

Boundary clipping

Ch6.1.6

Additional read (read first element)

Read once in each iteration

Reuse data using shift registers

Windowing of “1D” data stream - Result

Poor Arch

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
772	772	3.860 us	3.860 us	772	772	none

II = 3

Windowing

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
262	262	1.310 us	1.310 us	262	262	none

II = 1

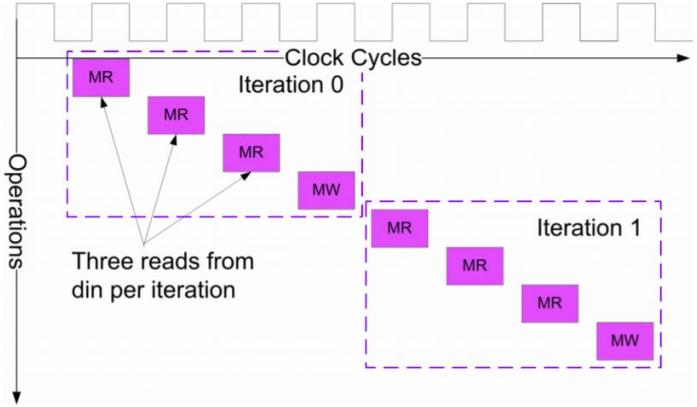


Illustration 116: Schedule for Moving Average with Poor Memory Architecture

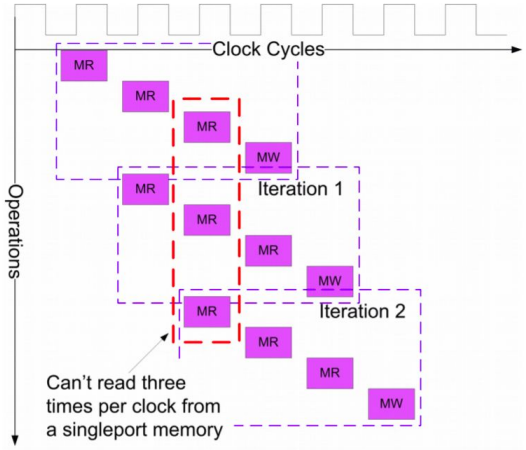


Illustration 117: Failed Schedule for Moving Average with II=1

Windowing of “1D” data stream - Result

Comparators

Poor Arch

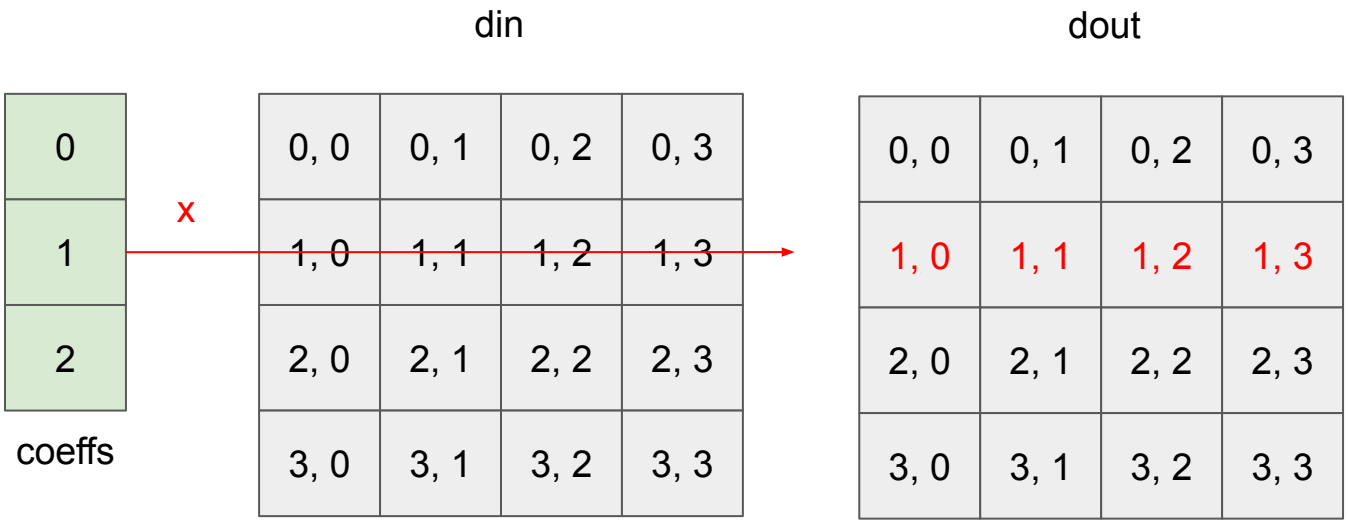
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	111	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	81	-
Register	-	-	101	-	-
Total	0	0	101	192	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	~0	~0	0

Windowing

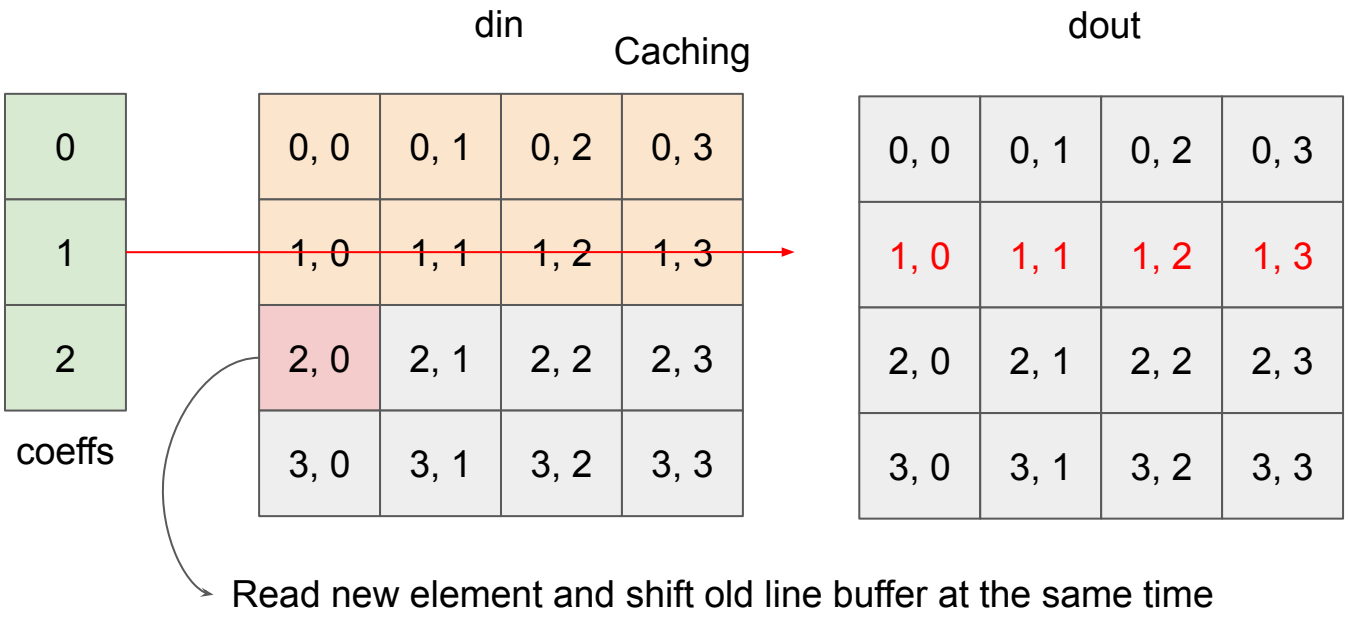
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	128	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	57	-
Register	0	-	226	64	-
Total	0	0	226	249	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	~0	~0	0

1. Comparators
2. Pipeline registers
3. Shift registers

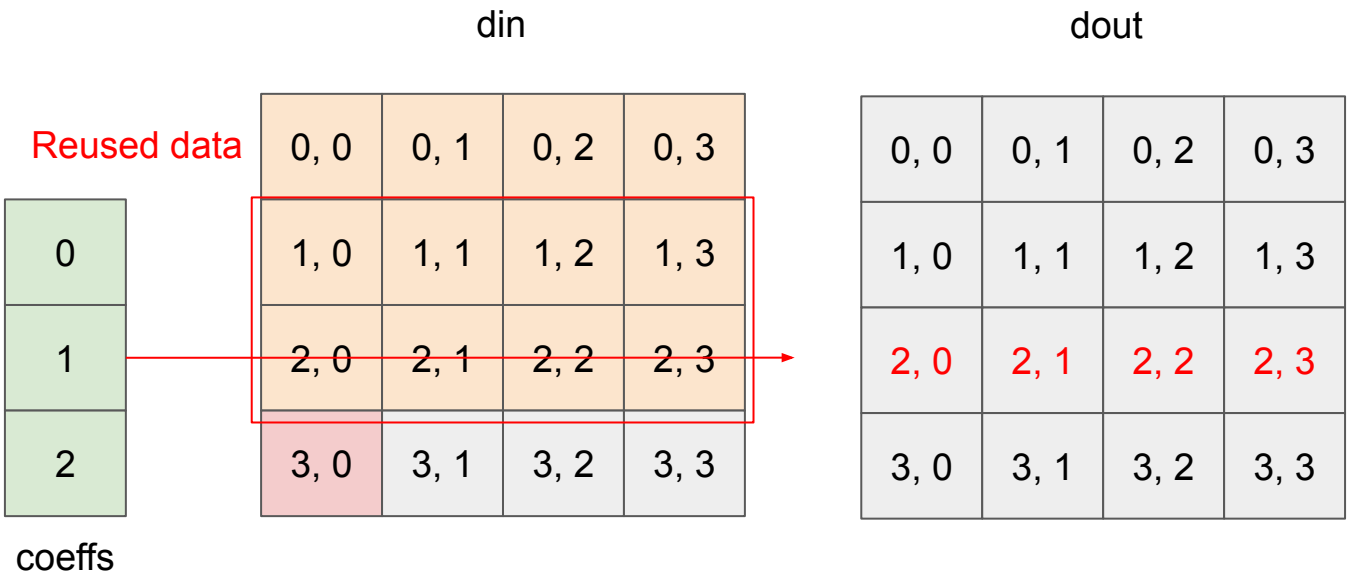
Windowing of “2D” data stream - Access Pattern



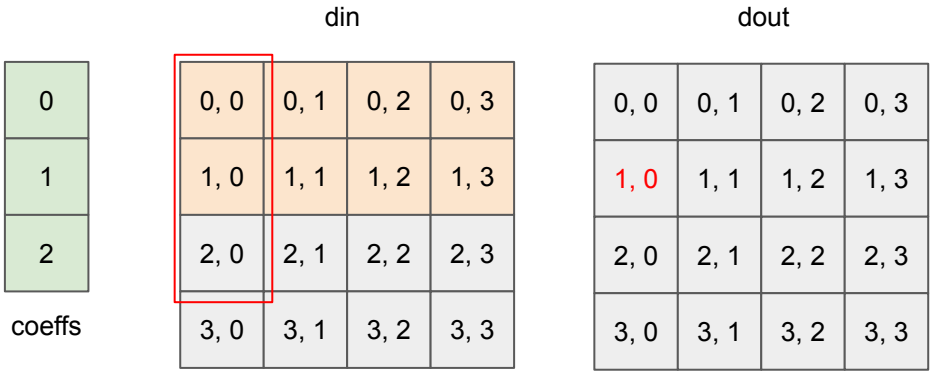
Windowing of “2D” data stream - Idea



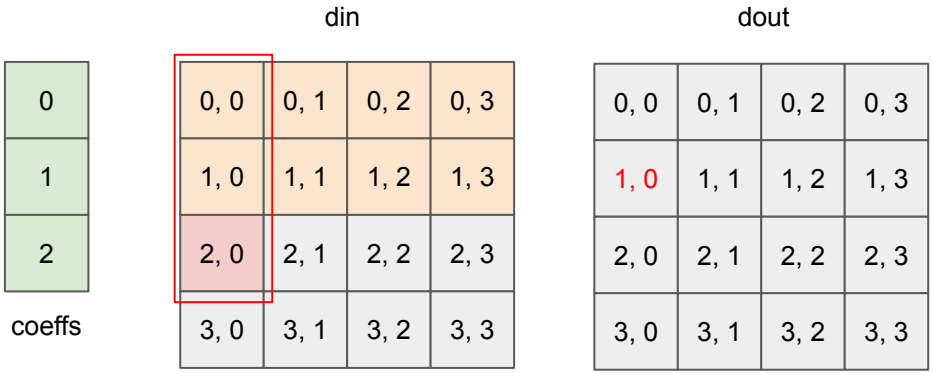
Windowing of “2D” data stream - Idea

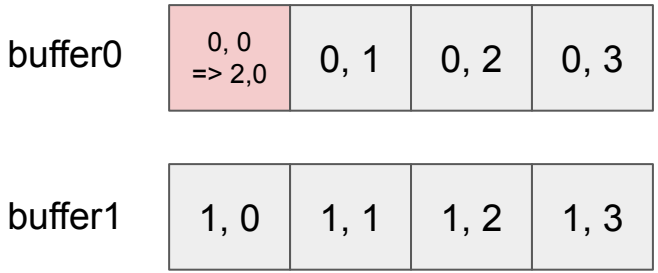


Windowing of “2D” data stream - Line Buffer



Windowing of “2D” data stream - Line Buffer





Windowing of “2D” data stream - Line Buffer

coeffs

0
1
2

din

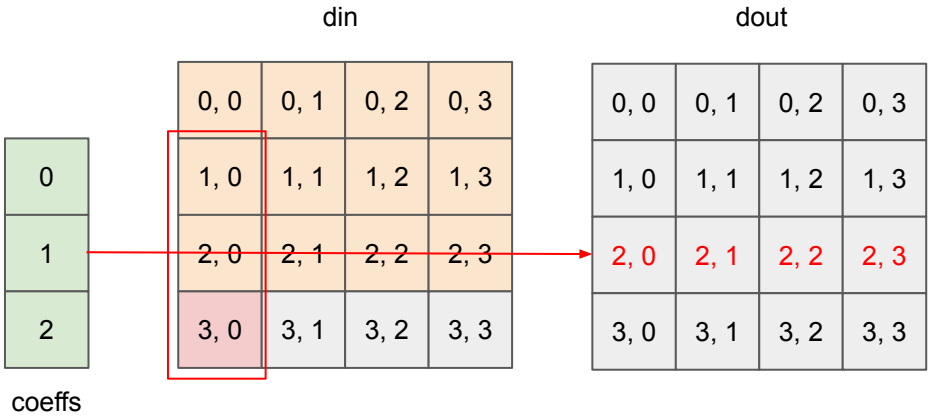
0, 0	0, 1	0, 2	0, 3
1, 0	1, 1	1, 2	1, 3
2, 0	2, 1	2, 2	2, 3
3, 0	3, 1	3, 2	3, 3

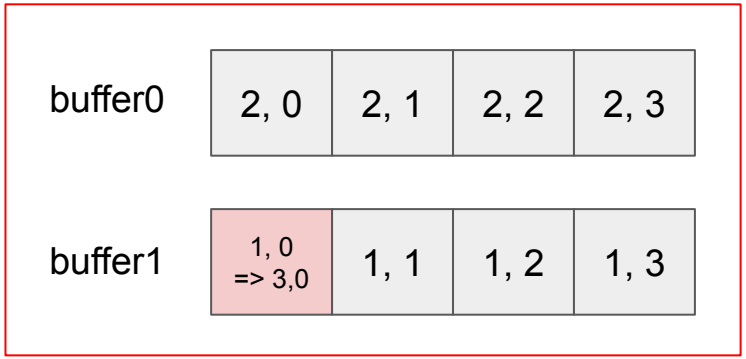
dout

0, 0	0, 1	0, 2	0, 3
1, 0	1, 1	1, 2	1, 3
2, 0	2, 1	2, 2	2, 3
3, 0	3, 1	3, 2	3, 3

buffer0	2, 0	0, 1 => 2,1	0, 2	0, 3
buffer1	1, 0	1, 1	1, 2	1, 3

Windowing of “2D” data stream - Line Buffer





Circular Buffer!

Problem: Impractical to implement line buffer using shift registers

Circular Buffer Implementation using RAM

- RAM with separate read/write ports require as much as 50% more area than a true singleport RAM
- The problem with using singleport RAM is that it cannot be read and written in the same clock cycle
  - Can not pipeline read / write operations
- Using coding technique to resolve this issue

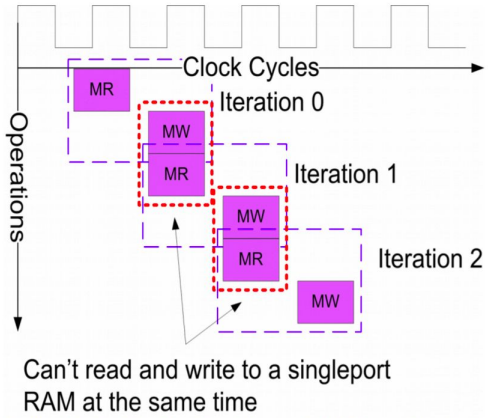
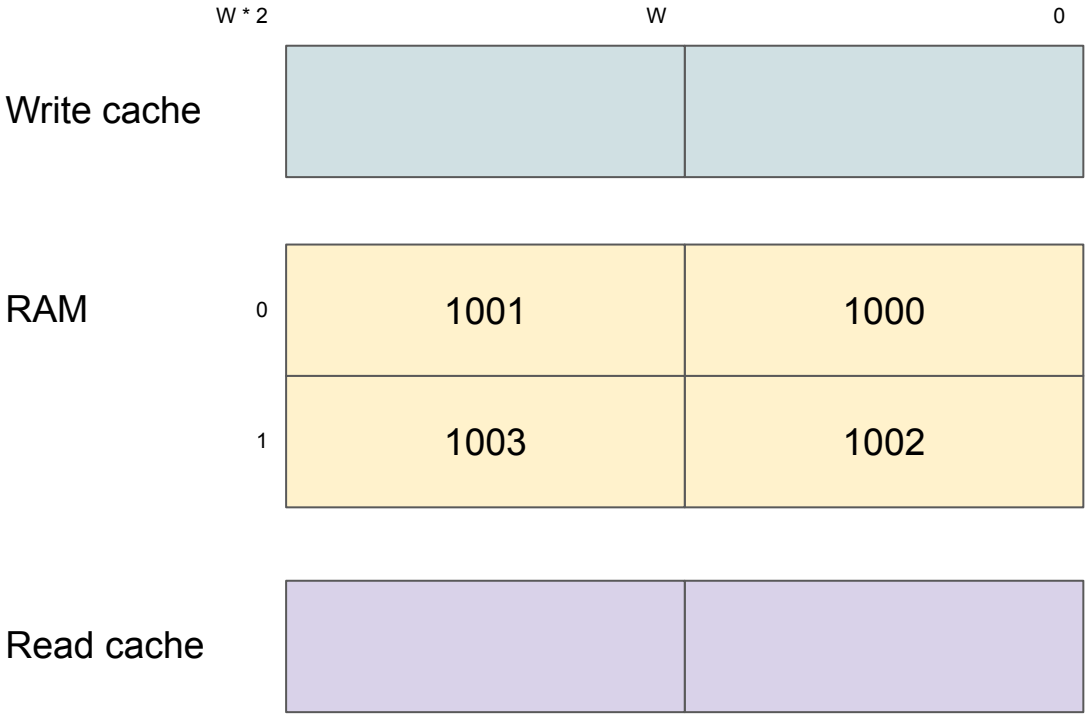
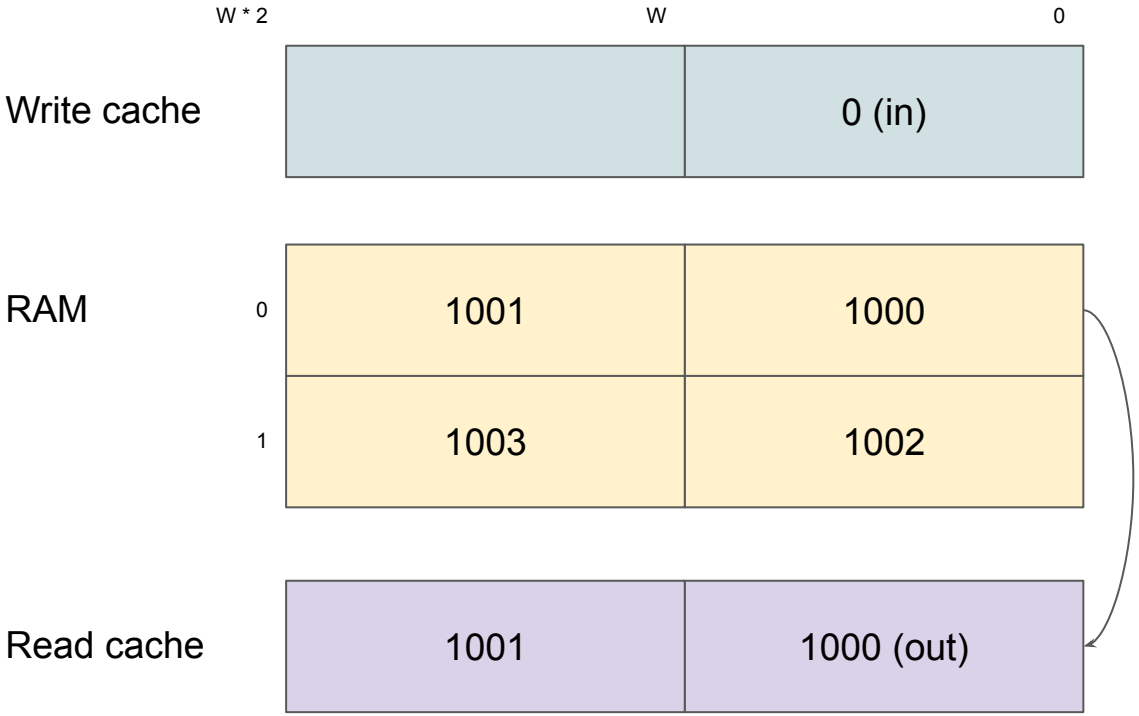


Illustration 115: Failed Schedule for Reading and Writing a Singleport RAM with II=1

Singleport RAM - How it works?

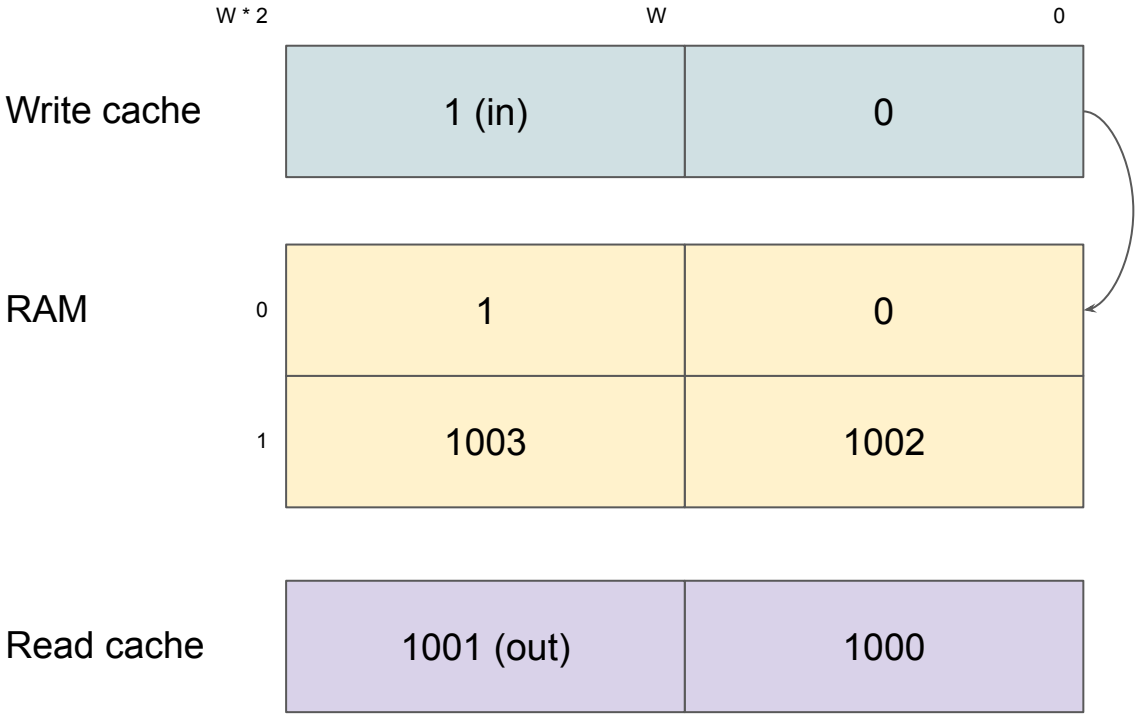


Singleport RAM - Iteration 0



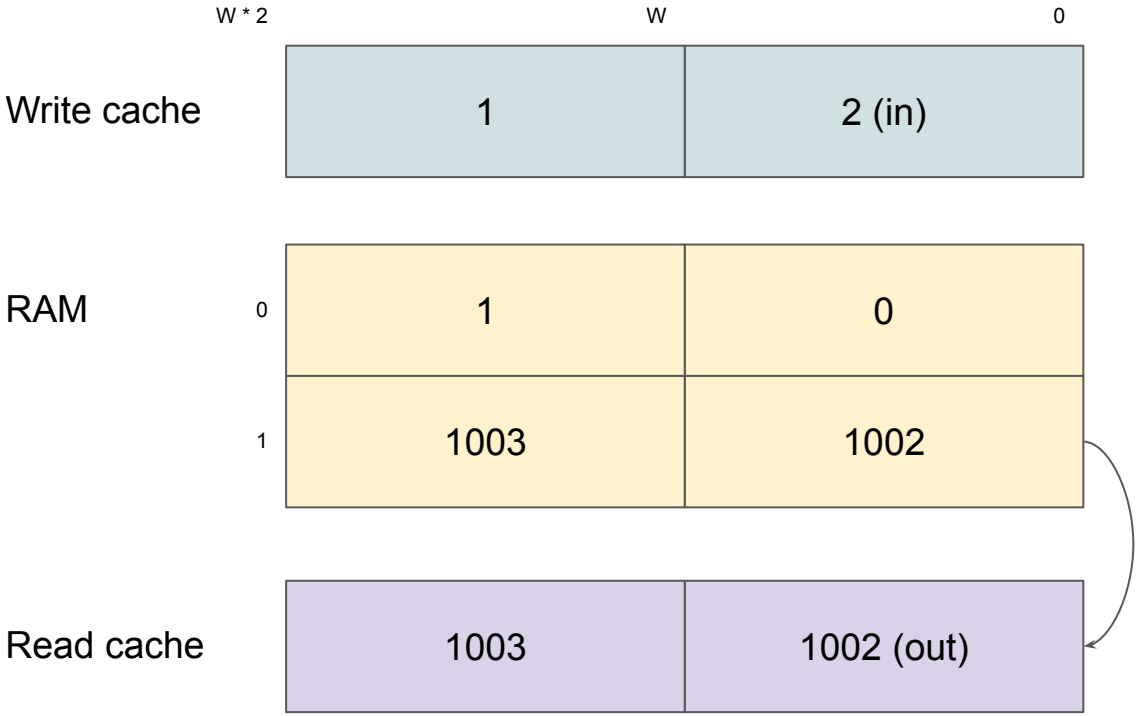
addr = 0 >> 1 = 0
=> read RAM[0]

Singleport RAM - Iteration 1



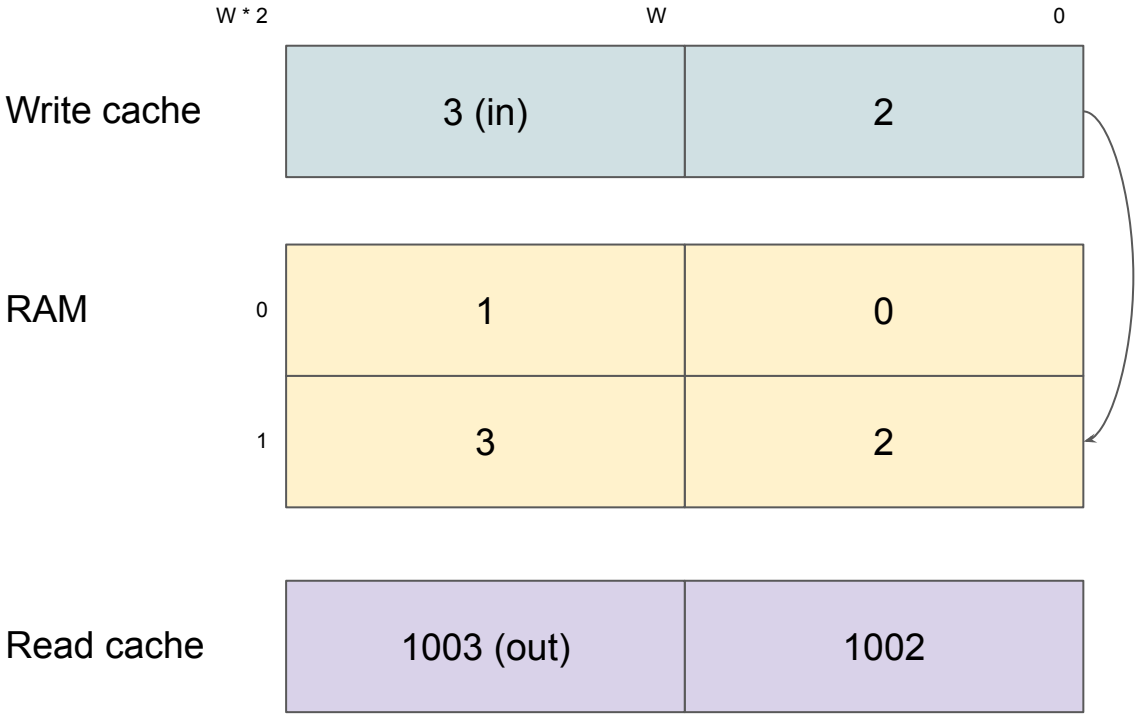
addr = 1 >> 1 = 0
=> write RAM[0]

Singleport RAM - Iteration 2



addr = 2 >> 1 = 1
=> read RAM[1]

Singleport RAM - Iteration 3



addr = 3 >> 1 = 1
=> write RAM[1]

Singleport RAM - Implementation

Widening width

```
// N: numbers of array elements
// W: word width
template <int N, int W>
class singleport_ram {
    // internal address
    int addr_int;
    // internal ram: width: W * 2, elements: N / 2 (assume N to be evenly divisable by two)
    ap_uint<W * 2> ram[N / 2];
    // single bit counter that is used to control reading and writing of data
    ap_uint<1> cnt;
    // internal caching: width: W * 2
    ap_uint<W * 2> read_data;
    ap_uint<W * 2> write_data;

   public:
    singleport_ram (): addr_int(0), cnt(0), read_data(0), write_data(0) {
#pragma HLS RESOURCE variable=ram core=RAM_1P_BRAM
    }
```

Singleport RAM - Implementation

```
ap_uint<W> exec(ap_uint<W> data_in, int addr, bool write) {
    ap_uint<W> tmp;
    addr_int = addr;
    // manipulate write cache
    if (write) {
        if (cnt == 0) { // write to lower halves
            write_data = (ap_uint<W * 2>(write_data.range(W * 2 - 1, W)) << W) | data_in;
        }
        else { // write to upper halves
            write_data = (ap_uint<W * 2>(data_in) << W) | ap_uint<W>(write_data.range(W - 1, 0));
        }
    }
    // control whether the internal array "ram" is read or written
    // addr_int >> 1 since number of elements is N / 2
    if (cnt == 0) {  //read on even
        read_data = ram[addr_int >> 1];
    }
    else {
        if (write) {
            ram[addr_int >> 1] = write_data;
        }
    }
    // read half of read cache
```

"cnt" decide which part to write

"cnt" decide which part to read

ram operation in current iteration

```
    // read half of read cache
    if (cnt == 0) {
        tmp = read_data.range(W - 1, 0);
    }
    else {
        tmp = read_data.range(W * 2 - 1, W);
    }
    cnt++;
    return tmp;
}
```

Windowing of “2D” data stream - Implementation

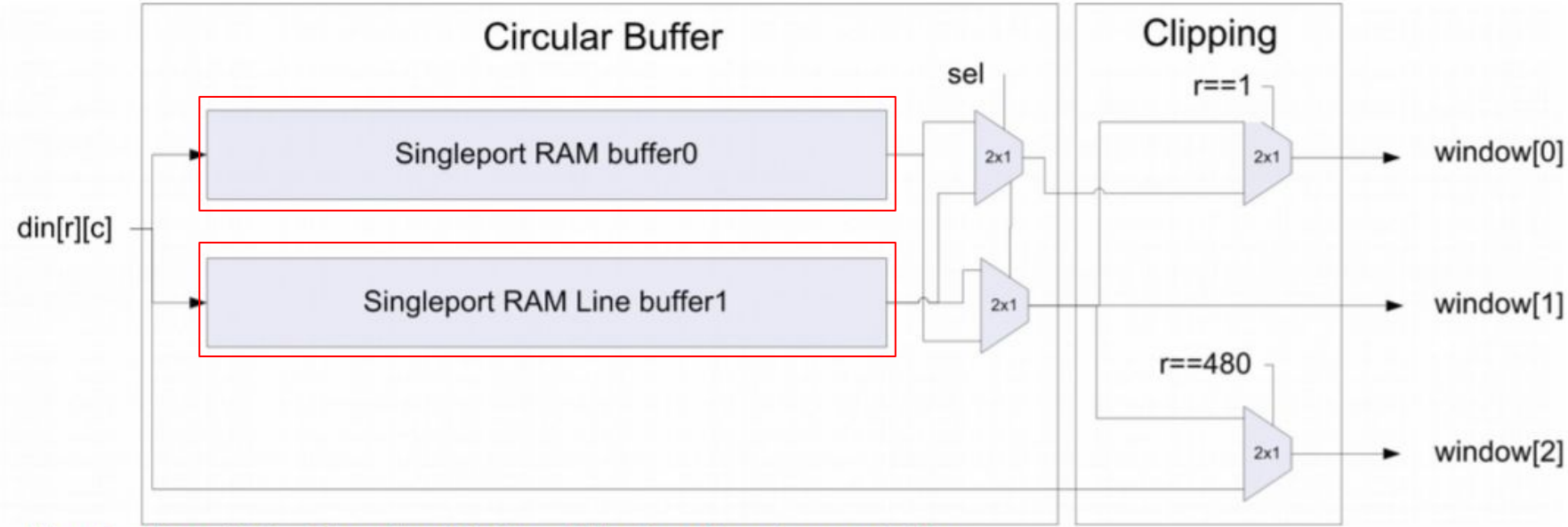
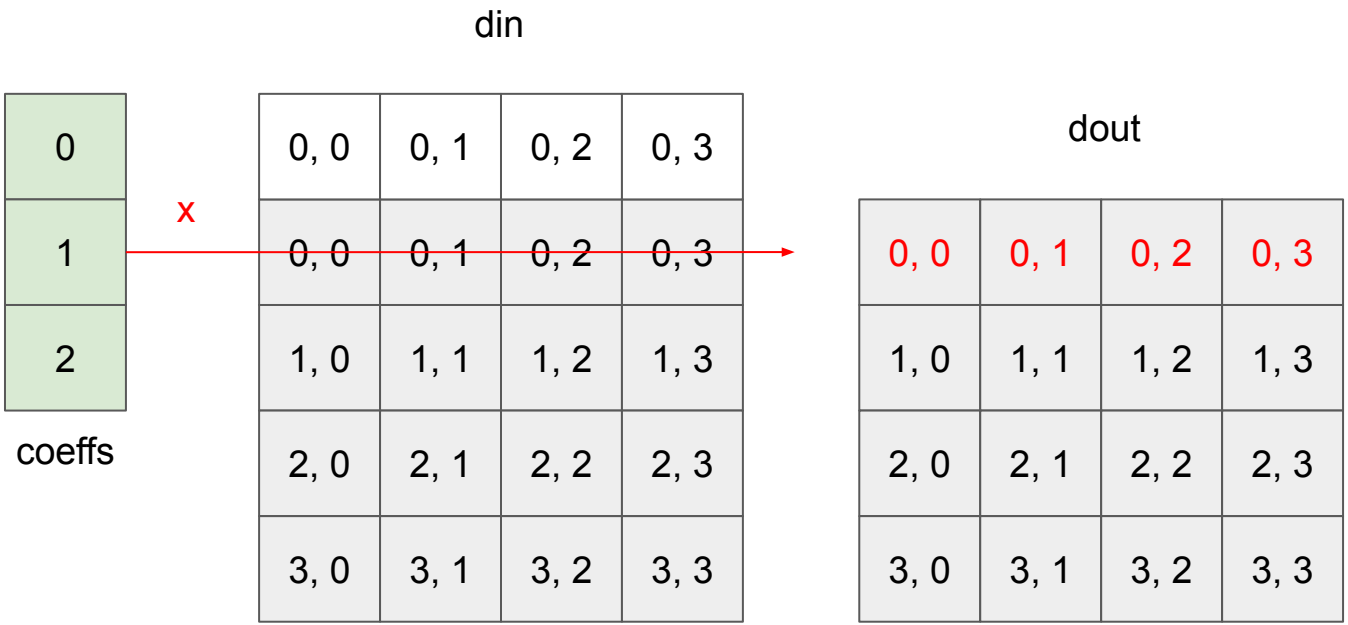


Illustration 121: Circular Buffer Window Implementation

Windowing of “2D” data stream - Boundary Conditions



Windowing of “2D” data stream - Poor Architecture

```
#include "ap_int.h"
#include "ap_fixed.h"
#include "window_2d.h"

int clip(int i) {
    int tmp = i;
    if (tmp < 0)
        tmp = 0;
    else if (tmp > NUM_ROW - 1)
        tmp = NUM_ROW - 1;
    return tmp;
}

void avg(ap_uint<8> din[NUM_ROW][NUM_COL], ap_uint<8> dout[NUM_ROW][NUM_COL]) {
#pragma HLS RESOURCE variable=din core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=dout core=RAM_1P_BRAM
    const ap_ufixed<3, 1> coeffs[3] = {0.25, 0.5, 0.25};
    ap_ufixed<13, 11> tmp;

ROW:
    for (int r = 0; r != NUM_ROW; r++) {
    COL:
        for (int c = 0; c != NUM_COL; c++) {
#pragma HLS PIPELINE II=3
            tmp = din[clip(r - 1)][c] * coeffs[0] + din[r][c] * coeffs[1] + din[clip(r + 1)][c] * coeffs[2];
            dout[r][c] = tmp.to_int();
        }
    }
}
```

Iterate row

Iterate column

Read1

Read2

Read3

Windowing of “2D” data stream - Implementation

```
void window_avg(ap_uint<8> din[NUM_ROW][NUM_COL], ap_uint<8> dout[NUM_ROW][NUM_COL]) {
#pragma HLS RESOURCE variable=din core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=dout core=RAM_1P_BRAM
    const ap_ufixed<3, 1> coeffs[3] = {0.25, 0.5, 0.25};
    ap_fixed<13, 11> tmp;
    ap_uint<8> w[3];
    ap_uint<8> din_tmp;
ROW:
    // first two rows must be read (r == 0 and r == 1) before there is
    // enough data to start computing the output
    for (int r = 0; r != NUM_ROW + 1; r++) {
    COL:
        for (int c = 0; c != NUM_COL; c++) {
#pragma HLS PIPELINE II=1
            if (r != NUM_ROW)
                din_tmp = din[r][c];
            buffer(din_tmp, c, w);
            clip_window(r, w);
            tmp = w[0] * coeffs[0] + w[1] * coeffs[1] + w[2] * coeffs[2];
            if (r != 0)
                dout[r - 1][c] = tmp.to_uint();
        }
    }
}
```

Additional iteration (read first row)

Buffer module

Clip module

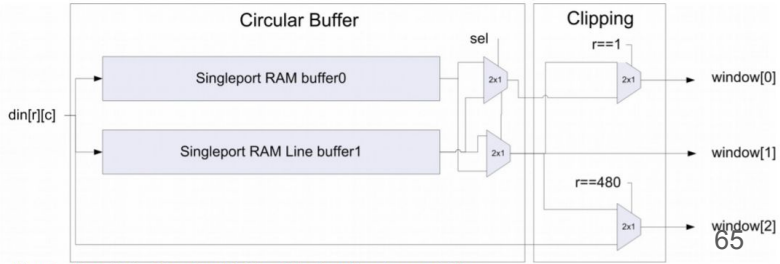


Illustration 121: Circular Buffer Window Implementation

Windowing of “2D” data stream - Implementation

```
void buffer(ap_uint<8> din, int c, ap_uint<8> window[3]) {
    static singleport_ram<720, 8> buffer0;
    static singleport_ram<720, 8> buffer1;
    ap_uint<8> t0, t1;
    // one bit counter that is used to select between the two line buffers
    static ap_uint<1> sel = 1;
    // check current address or column position "c" and increments "sel" at the
    // start of a new row
    if (c == 0)  //switch buffer write at start of line
        sel += 1;
    // The input data "din" is passed to both memories along with the address "c"
    // "sel" is only active for one memory at a time so only one of the memories is written.
    t1 = buffer1.exec(din, c, sel);
    t0 = buffer0.exec(din, c, !sel);

    window[0] = (sel == 1) ? t1 : t0;
    window[1] = (sel == 1) ? t0 : t1;
    window[2] = din;
}
```

One buffer is read-only

When new line is acquired (c = 0), change the line buffer that is written

Boundary clipping

```
void clip_window(int r, ap_uint<8> window[3]) {
    ap_uint<8> w[3];
    for (int i = 0; i < 3; i++)
        w[i] = window[i];
    window[0] = (r == 1) ? w[1] : w[0];
    window[1] = w[1];
    window[2] = (r == 480) ? w[1] : w[2];
}
```

Windowing of “2D” data stream - Result

Poor Arch

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1036807	1036807	5.184 ms	5.184 ms	1036807	1036807	none

II = 3

row: 480
col: 720

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- ROW_COL	1036805	1036805	9	3	1	345600	yes

Windowing

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
346333	346333	1.732 ms	1.732 ms	346333	346333	none

II = 1

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- ROW_COL	346331	346331	13	1	1	346320	yes

Windowing of “2D” data stream - Result

Poor Arch

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	3	-	-	-
Expression	-	-	0	265	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	117	-
Register	0	-	333	32	-
Total	0	3	333	414	0
Available	280	220	106400	53200	0
Utilization (%)	0	1	~0	~0	0

Windowing

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	2	-	-	-
Expression	-	-	0	238	-
FIFO	-	-	-	-	-
Instance	2	-	424	312	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	75	-
Register	0	-	689	256	-
Total	2	2	1113	881	0
Available	280	220	106400	53200	0
Utilization (%)	~0	~0	1	1	0

1. ap_enable_iterX
2. icmp
3. select

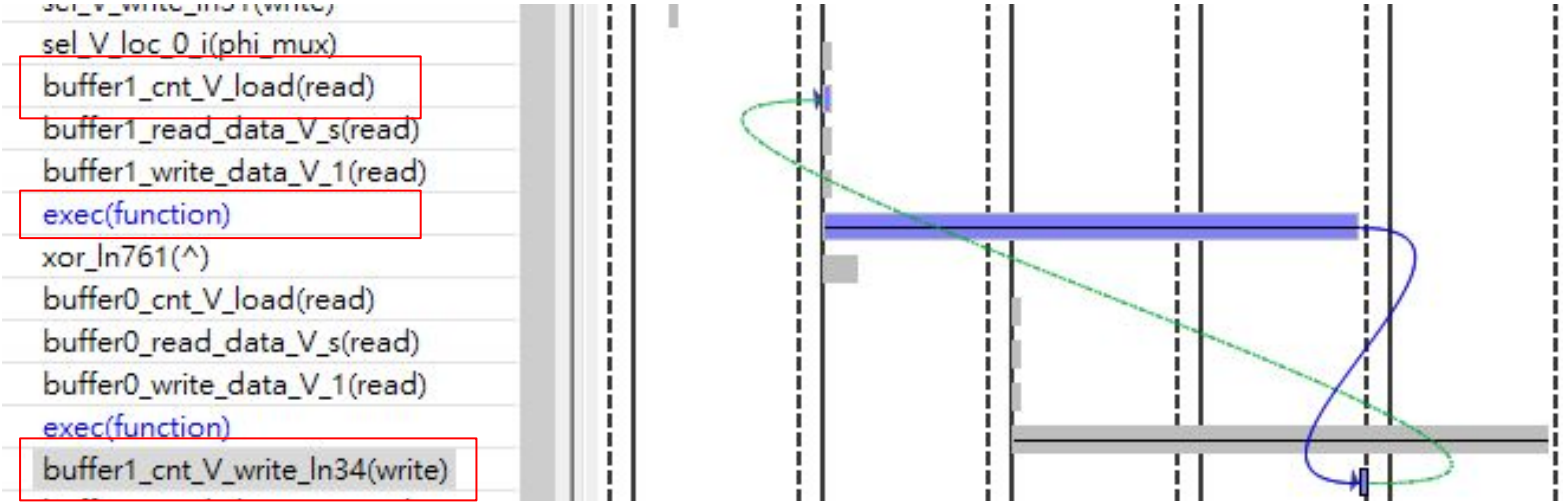
icmp_ln13_reg_460	64	32	1	0
icmp_ln52_reg_405	64	32	1	0
icmp_ln54_reg_414	64	32	1	0
icmp_ln61_1_reg_443	64	32	1	0
select_ln52_1_reg_470	64	32	1	0
select_ln52_2_reg_476	64	32	1	0
select_ln52_3_reg_449	64	32	9	0
select_ln52_reg_421	64	32	10	0

Problem we met

singleport's exec function is synthesized as a seperate module

c-sim success, co-sim failed

```
WARNING: [SCHED 204-68] The II Violation in module 'window_avg' (Loop: ROW_COL):
      Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 0)
   between 'store' operation ('buffer1_cnt_V_write_ln34',...) of variable 'newret2',
                                                          ... on static variable 'buffer1_cnt_V'
       and 'load' operation ('buffer1_cnt_V_load', ...) on static variable 'buffer1_cnt_V'.
```

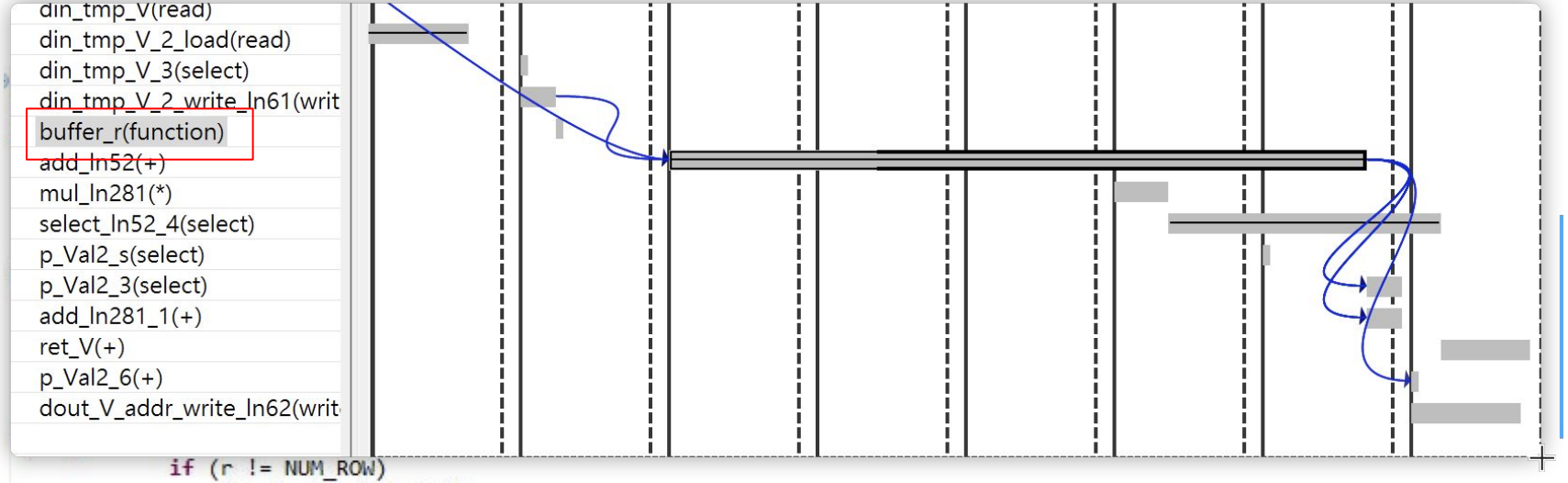


Problem we met - Solution

```
    ap_uint<W> exec(ap_uint<W> data_in, int addr, bool write) {
#pragma HLS INLINE
        ap_uint<W> tmp;
        addr_int = addr;
        // manipulate write cache
        if (write) {
            if (cnt == 0) { // write to lower halves
                write_data = (ap_uint<W * 2>(write_data.range(W * 2 - 1, W)) << W) | data_in;
            }
            else { // write to upper halves
                write_data = (ap_uint<W * 2>(data_in) << W) | ap_uint<W>(write_data.range(W - 1, 0));
            }
        }
```

Inlining to parent function

Problem we met - Solution




```
        if (r != NUM_ROW)
            din_tmp = din[r][c];
        buffer(din_tmp, c, w);
        clip_window(r, w);
        tmp = w[0] * coeffs[0] + w[1] * coeffs[1] + w[2] * coeffs[2];
        if (r != 0)
            dout[r - 1][c] = tmp.to_uint();
    }
  }
}
```

Outline

- 7.1. Memory-based Shift Register
- 7.2. Memory Organization
- 7.3. Widening the Word Width of Memories
- 7.4. Caching
- Achieving Multi-port memory performance on Single-port memory with coding technique

Achieving Multi-Port Memory Performance on Single-Port Memory with Coding Techniques

Hardik Jain
Department of ECE
The University of Texas at Austin
Austin, United States
hardikbjain@utexas.edu

Matthew Edwards
GenXComm Inc.
Austin, United States
matthewedwards@genxcomminc.com

Ethan Elenberg
ASAPP Inc.
New York,[1] United States
eelenberg@asapp.com

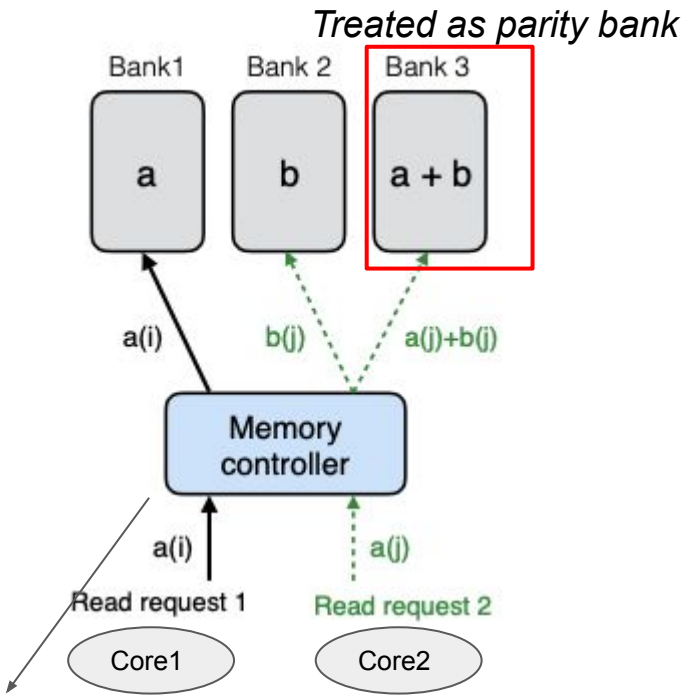
Ankit Singh Rawat
Google
New York[1], United States
ankitsrwt@gmail.com

Sriram Vishwanath
Department of ECE
University of Texas at Austin
Austin, United States
sriram@utexas.edu

International Conference on Information and Computer Technologies (ICICT) 2020

Quick Start

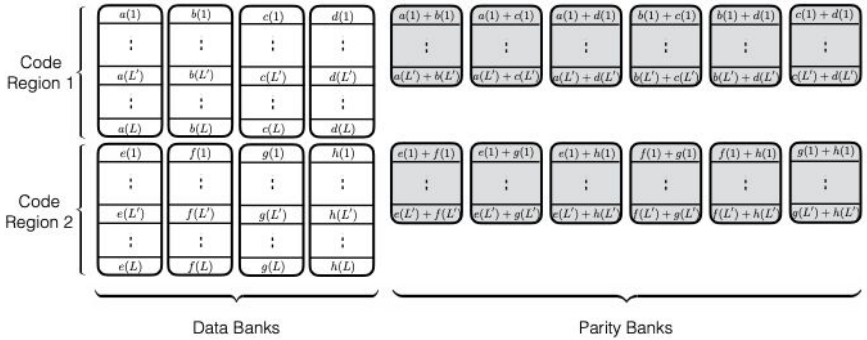
- We only have **single port** (read / write each) for each bank
- If we want to read data in the same bank, say **a[1]** and **a[2]**
  - First, we should have a **parity bank**
  - Second, need preprocessing beforehand(**xor computation**)
- How?
  - Access a[1] directly from **bank-1**
  - Access a[2] by **bank-2 and bank-3**, computing by xor operation



Need to design carefully, and it is usually complicated

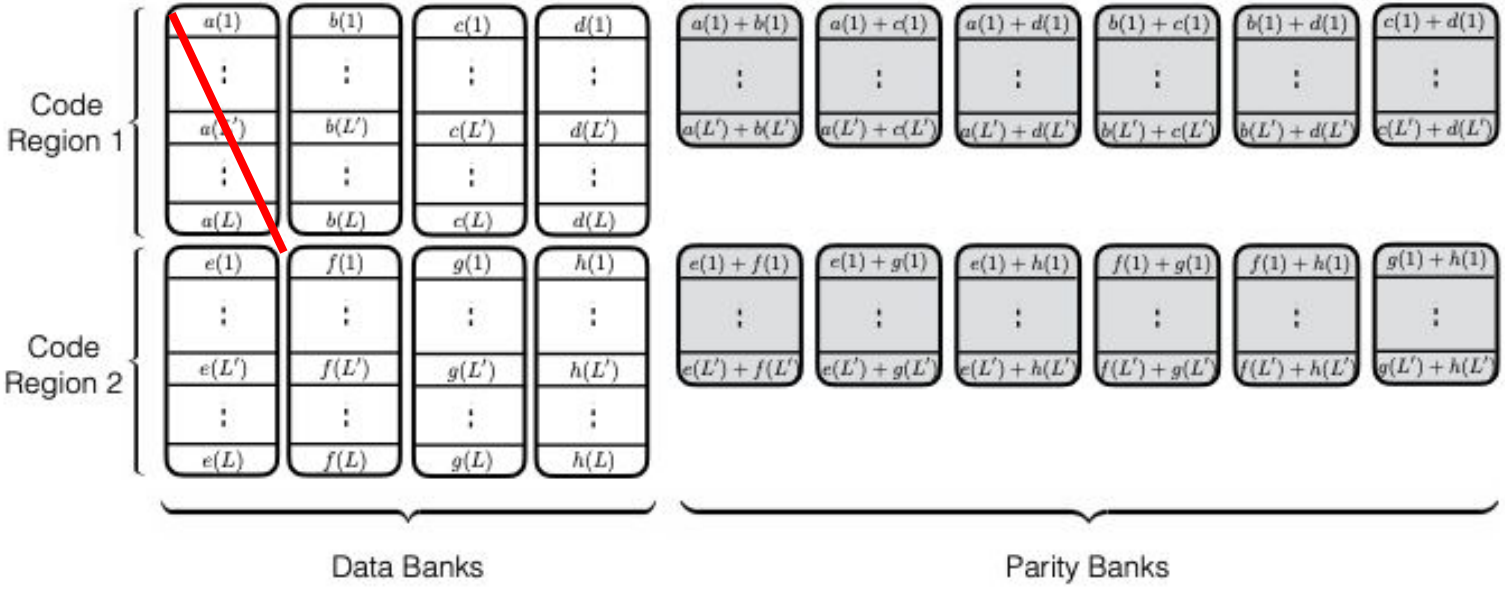
Code Scheme I

- Best Case:
  - Can achieve up to **10** parallel accesses to a particular coded region in one access cycle.
  - {a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), c(3), d(3)}
- Worst Case:
  - The worst case number of reads per cycle is equal to **the number of data banks.**
  - **No same row index**
  - {a(1), a(2), b(8), b(9), c(10), c(11), d(14), d(15)}



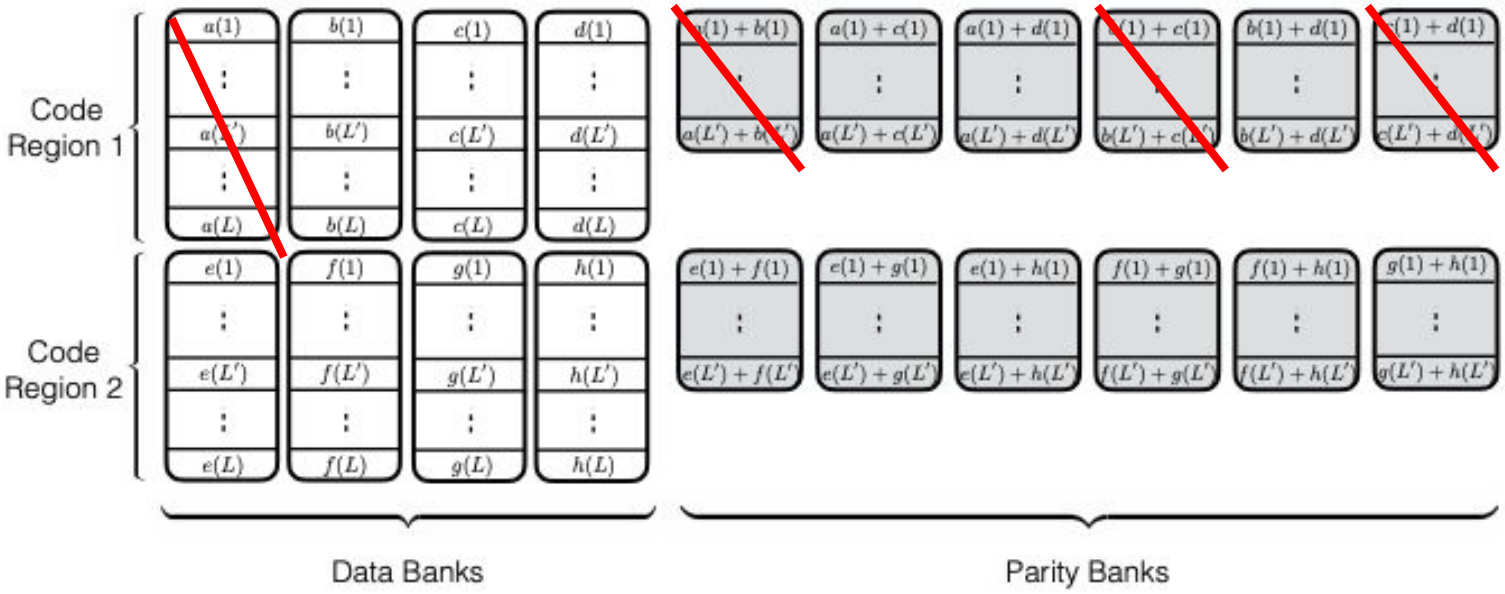
Code Scheme I (illustration)

{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), c(3), d(3)}



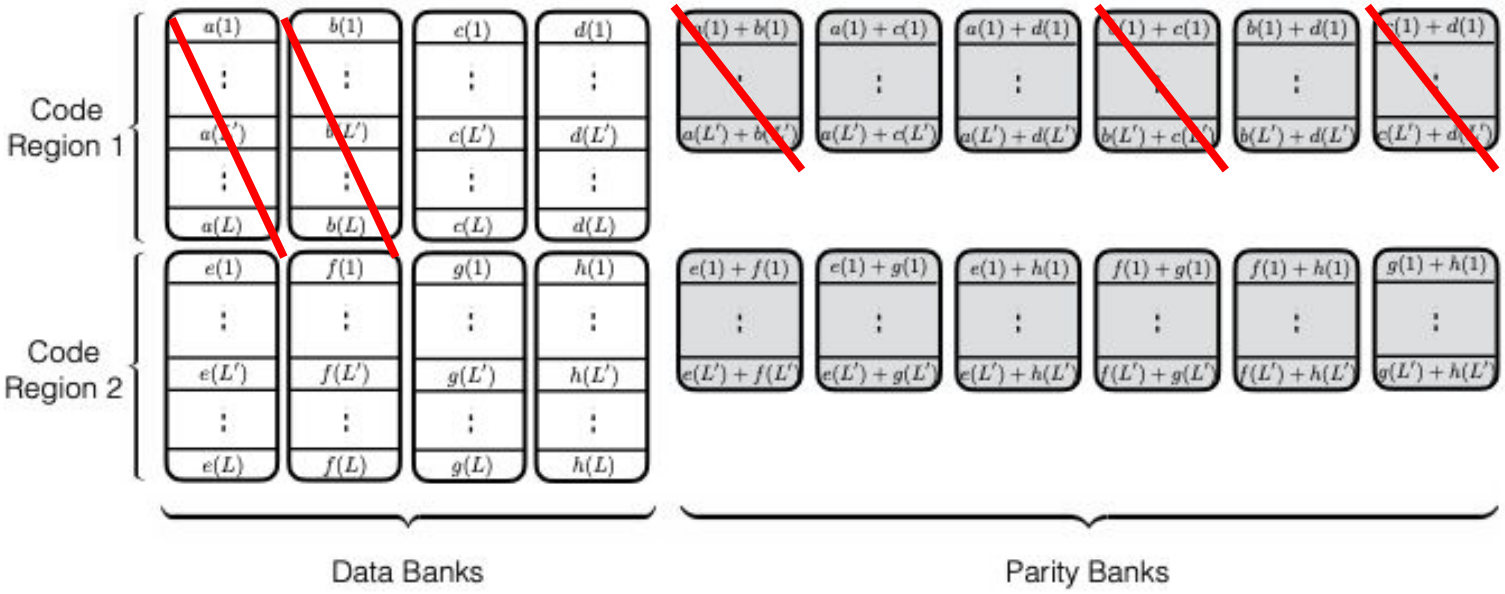
Code Scheme I (illustration)

{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), c(3), d(3)}



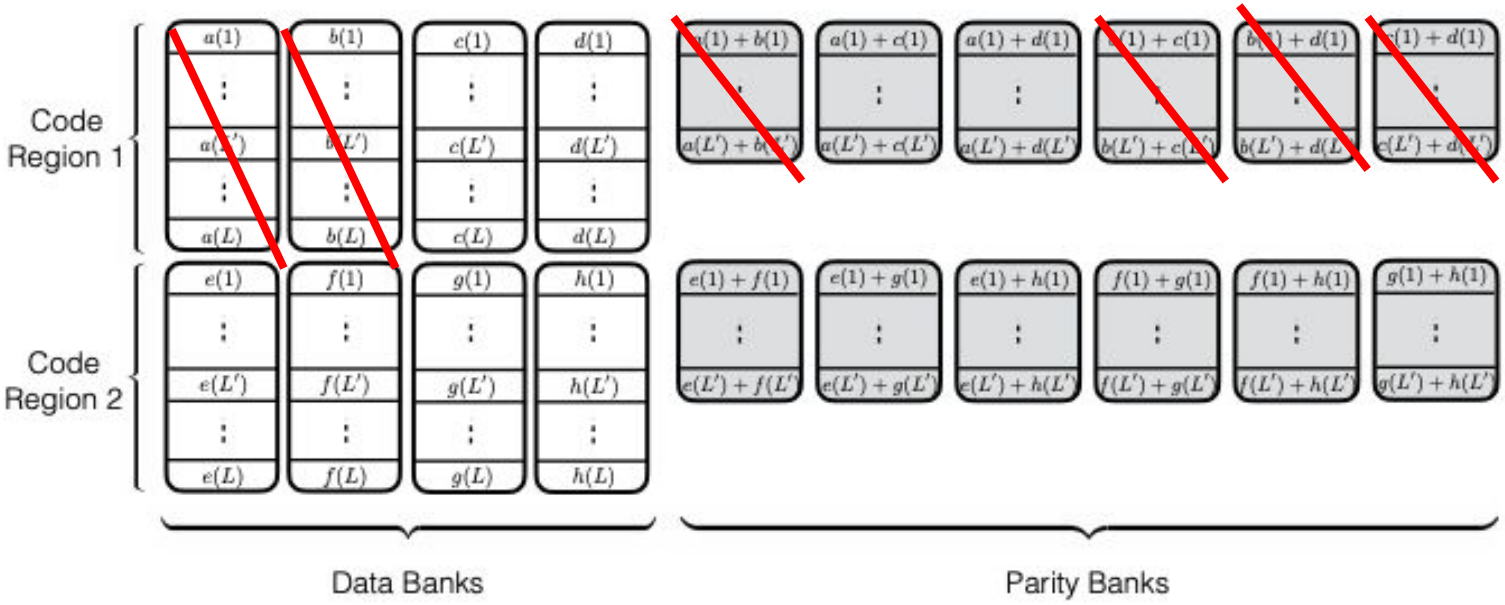
Code Scheme I (illustration)

{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), c(3), d(3)}



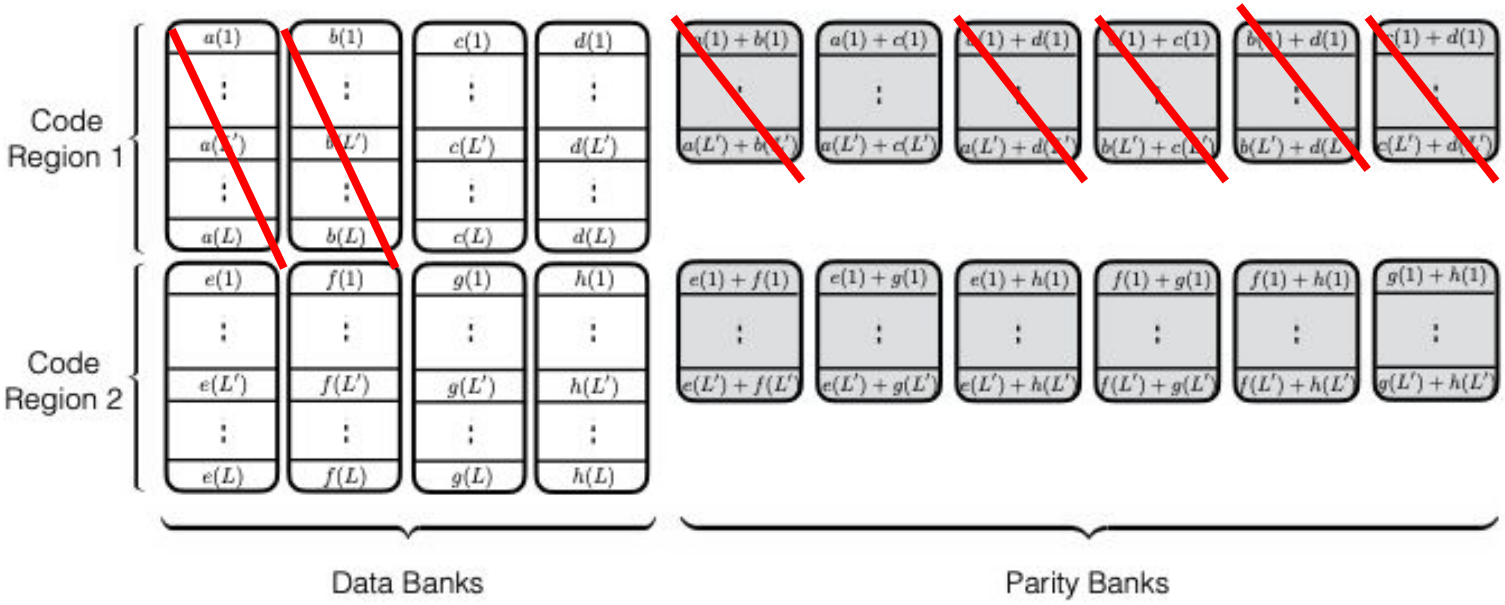
Code Scheme I (illustration)

{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), c(3), d(3)}



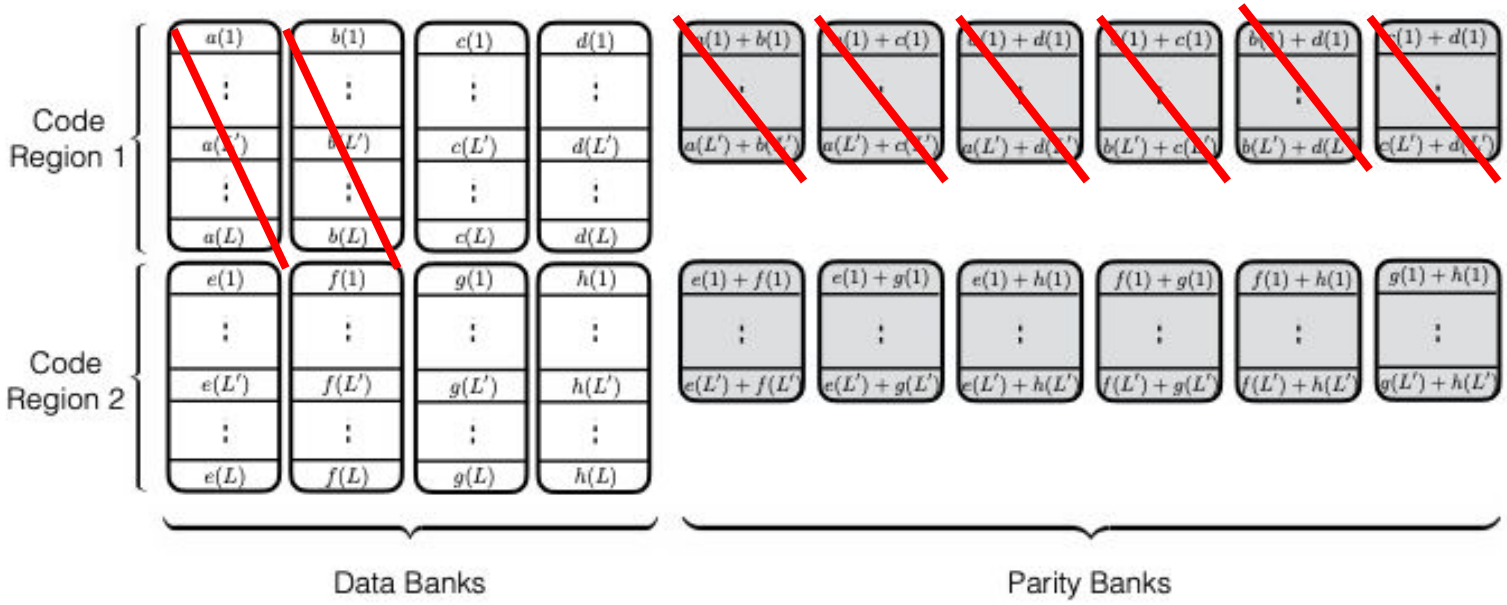
Code Scheme I (illustration)

{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), c(3), d(3)}



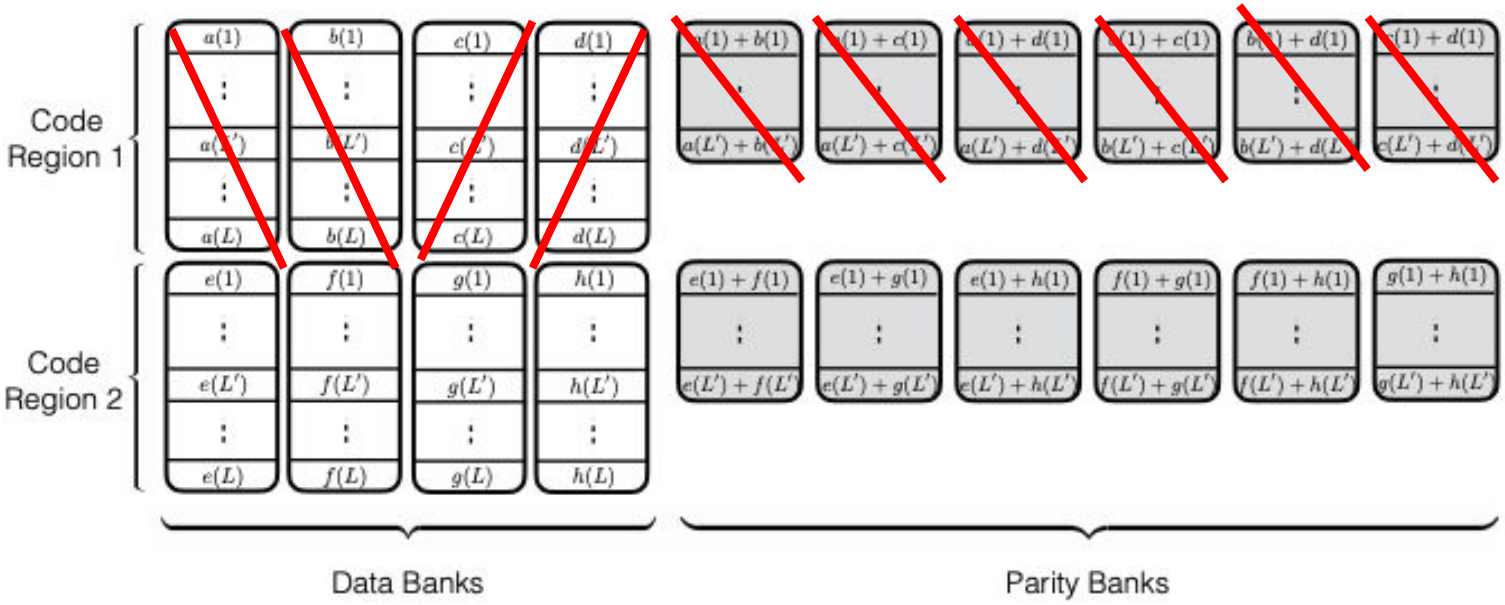
Code Scheme I (illustration)

{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), c(3), d(3)}



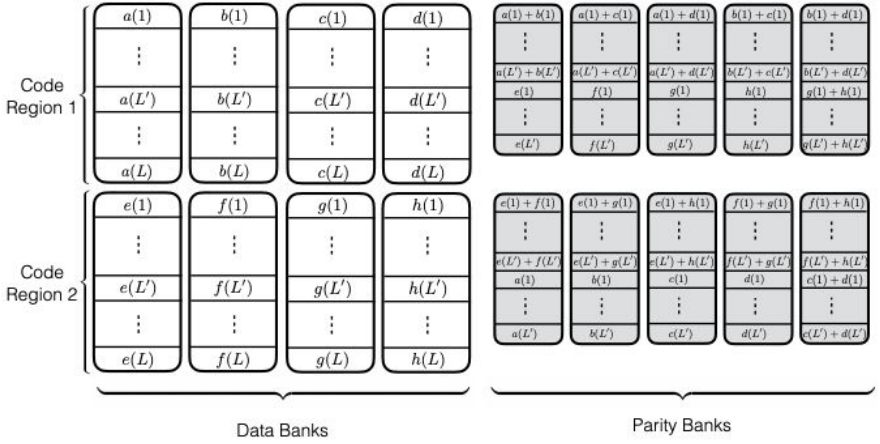
Code Scheme I (illustration)

{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), c(3), d(3)}



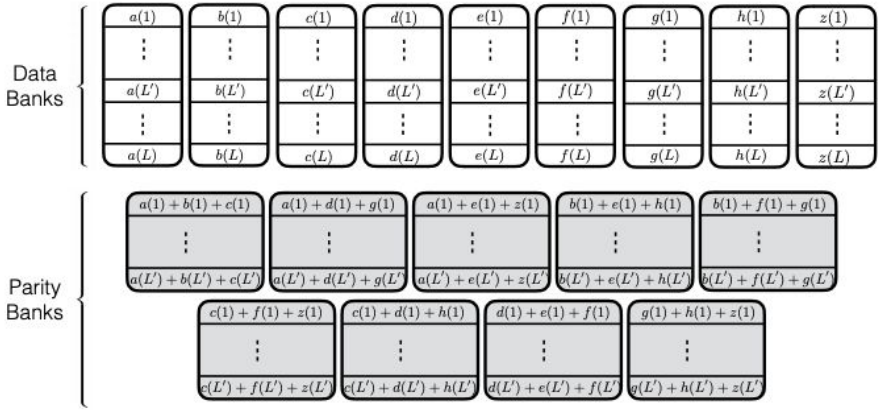
Code Scheme II

- Best Case:
  - Can support up to **9** read requests in a single memory clock cycle.
  - E.g. {a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), a(3), b(3), c(3)}
- Worst Case:
  - 5 simultaneous accesses in a single memory clock cycle in the worst case.



Code Scheme III

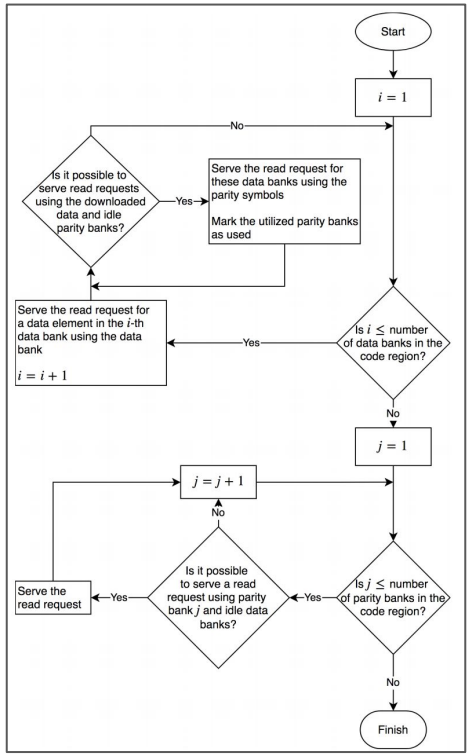
- Best Case:
  - Best case number of reads per cycle will be equal to **the number of data and parity banks.**
- Worst Case:
  - The number of reads per cycle is equal to the number of data banks.



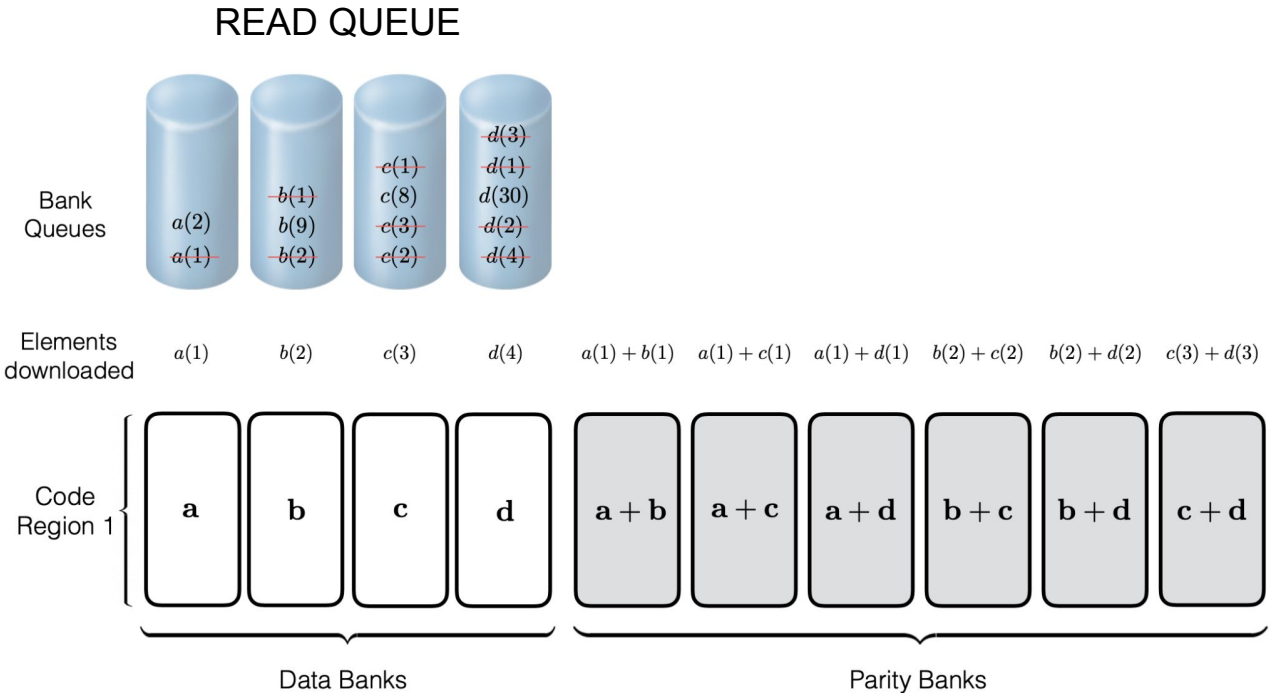
Memory Controller

- Core arbiter:
  - Receives **up to one request** from each core which it stores in an internal queue.
  - **Push** these requests to the **appropriate bank queue**.
  - If destination bank queue is **full**, then **stalls the core**.
- Bank queues:
  - Each data bank has a **corresponding read queue** and **write queue**.
  - The core arbiter sends memory requests to the bank queues until the queues are full.
- Access scheduler:
  - Every memory cycle, the **access scheduler(called pattern builder) chooses to serve read requests or write requests**, algorithmically determining which requests in the bank queues it will schedule.

Read Pattern Builder



Flowchart



Summary

- Implement Mentor HLS Chapter 7 (Memory Architectures) through Xilinx HLS
- Explore Xilinx HLS pragma for memory optimization
  - ARRAY_PARTITION (interleave)
  - ARRAY_RESHAPE (word widthening)
  - PIPELINE
  - INLINE
  - RESOURCE
- Explore application dataflow and design memory-friendly access pattern
- How to achieve multi-port ram performance through single-port ram
- Source code
  - https://github.com/kaiiiz/hls-bluebook-memory-architectures