

Lab A - Mentor HLS Bluebook

Scheduling of IO and Memories

HLS Team 05

游子緒 陳昱行 馬健凱

2021 / 04 / 13

GitHub repository: https://github.com/ChienKaiMa/2021_ACA_HLS_team05

Unconditional I/O

Mentor's Example

Accumulator with threshold

- Input
 - `int din[4]`
 - `int& / int` threshold
 - `bool& / bool` flag
- Output
 - `int& dout`

Accumulator with threshold

Case 1

- `int din[4] = {1, -10, 6, -7}`
- `int& / int threshold = 1`
- `bool& / bool flag = 0`
- `int& dout = -10`

Case 2

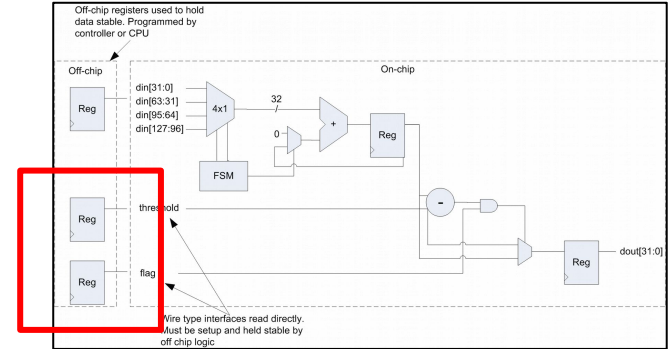
- `int din[4] = {13, -7, 6, 0}`
- `int& / int threshold = 7`
- `bool& / bool flag = 1`
- `int& dout = 6`

Statement of the HLS bluebook

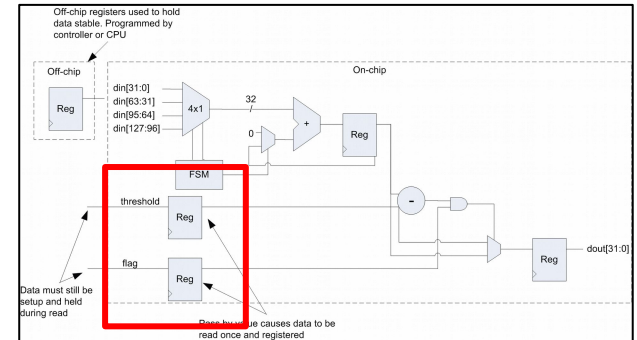
- Unconditional I/O
 - mapped to “wire”
 - no handshaking protocols
- Common use cases
 - control type interfaces
 - designs that are pipelined with $II=1$
(I/O is read or written every clock cycle)

Statement of the HLS bluebook

- (Uncond) Pass by reference
 - data is “off-chip”
 - **or** data is expected every clock cycle
- (Uncond) Pass by value
 - The data is read once
=> I/O traffic reduced
- Setup: ACCUM loop with $ll = 1$

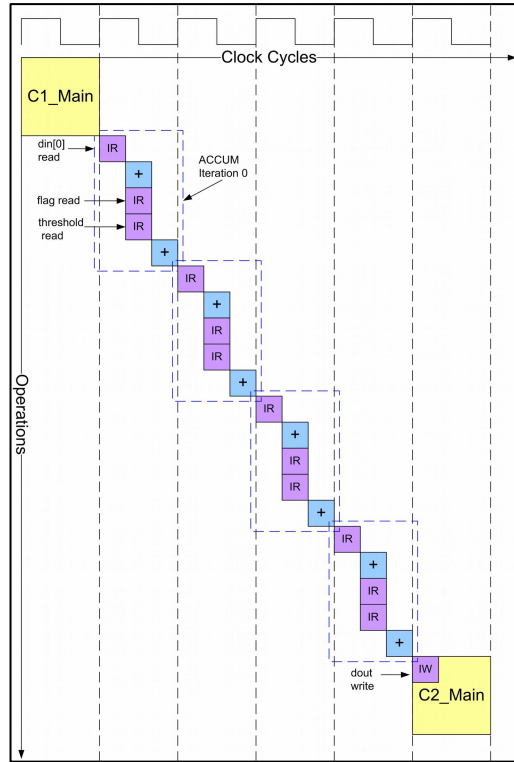


Ref: HLS Bluebook p.81 Illustration 63

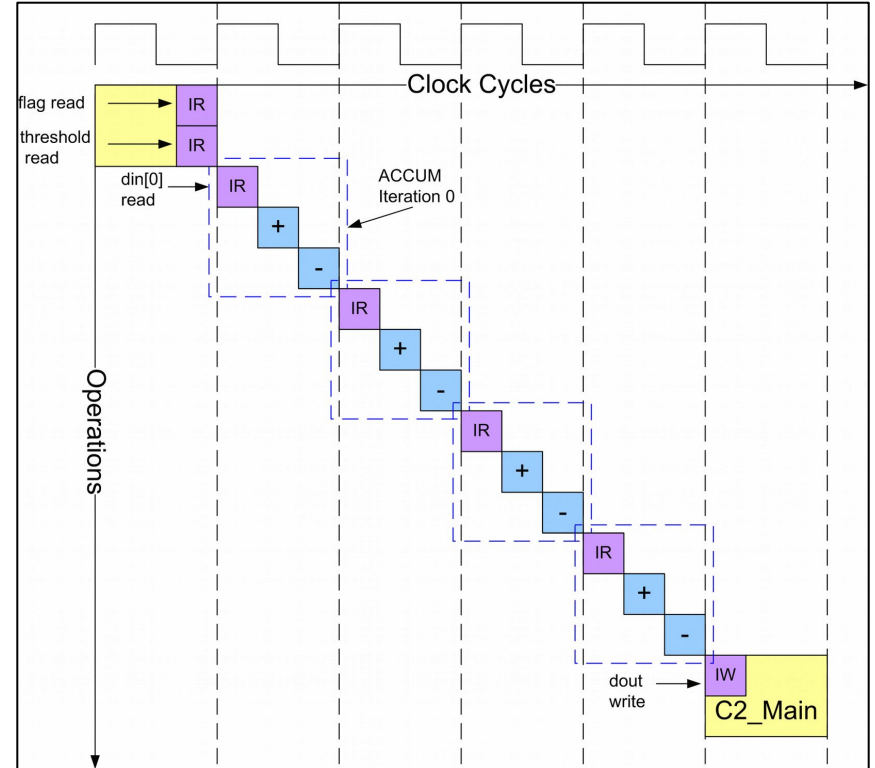


Ref: HLS Bluebook p.81 Illustration 65

Expected results



Ref: [HLS Bluebook p.81 Illustration 62](#)



Ref: [HLS Bluebook p.81 Illustration 64](#)

Directives for Unconditional I/O

```
#pragma HLS INTERFACE ap_fifo port=din
#pragma HLS INTERFACE ap_none port=flag
#pragma HLS INTERFACE ap_none port=threshold
#pragma HLS INTERFACE ap_none port=dout
#pragma HLS PIPELINE II=1
```

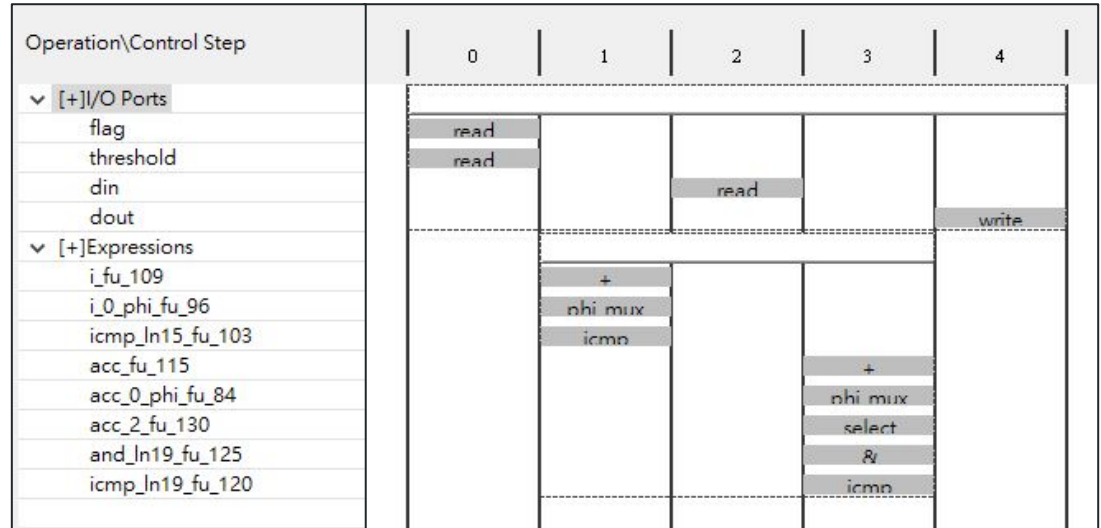
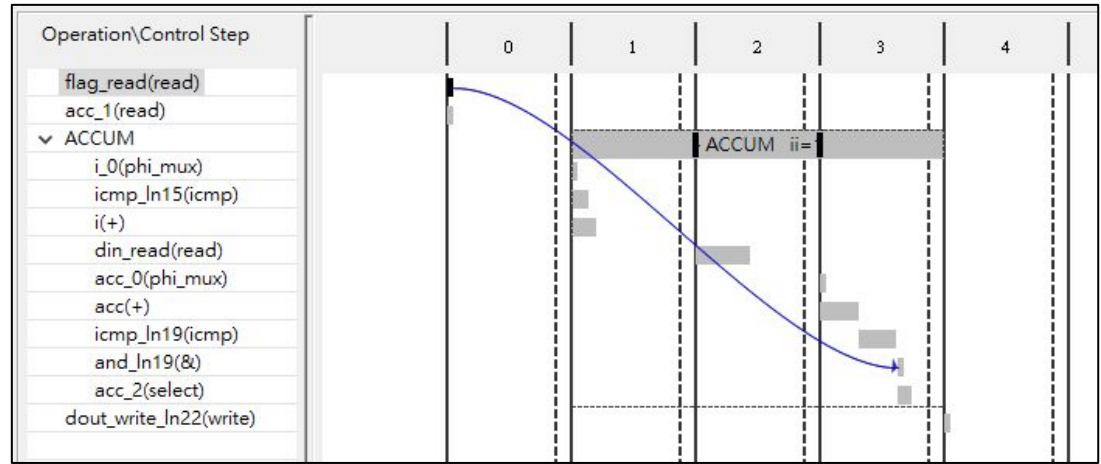
We use the same
pragma for both cases!

Results

Schedule

Both cases have the same result!

Resource



Conditional I/O




Statement of the HLS bluebook

- Conditional I/O
 - Has handshake
- (Cond) Pass by reference
 - threshold, flag ready for data
 - No need for ack
- (Cond) Pass by value
 - May have unexpected behavior
- Setup: ACCUM loop with $II = 1$

Vivado HLS Interface

Different from
Mentor's Catapult C

Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld								D	
ap_ovld							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

 Supported
  D = Default Interface
  Not Supported

X14293

Figure 1-39: Data Type and Interface Synthesis Support

Directives for Cond Pass by Ref.

```
#pragma HLS INTERFACE ap_fifo port=din
```

```
#pragma HLS INTERFACE ap_none port=flag
```

```
#pragma HLS INTERFACE ap_hs port=threshold
```

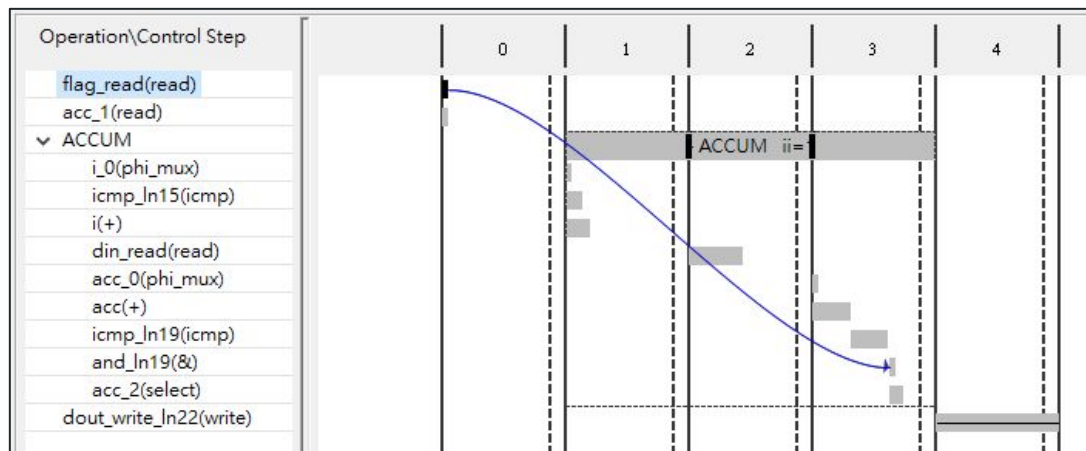
```
#pragma HLS INTERFACE ap_hs port=dout
```

```
#pragma HLS PIPELINE II=1
```

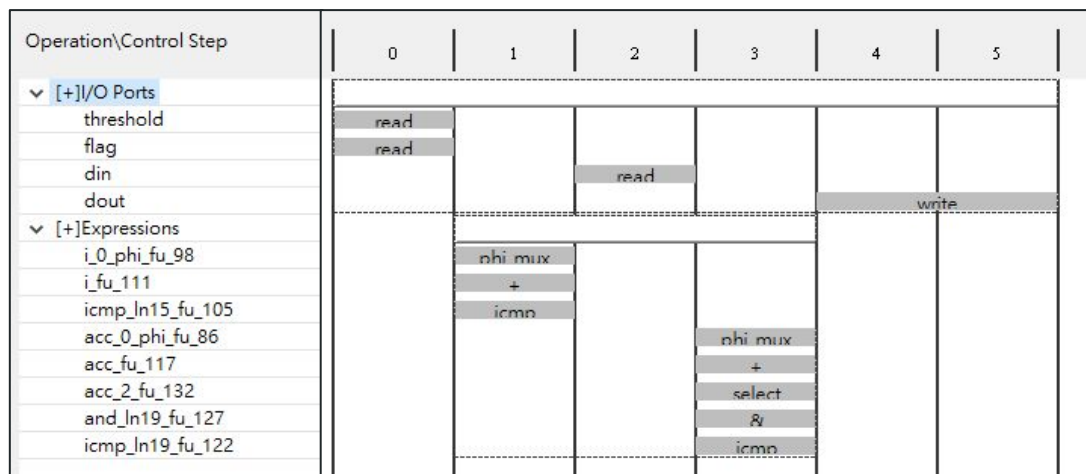
threshold, data mapped to ready to send or receive data interface

Results

Schedule



Resource



Directives for Cond Pass by Value

```
#pragma HLS INTERFACE ap_fifo port=din
```

```
#pragma HLS INTERFACE ap_none port=flag
```

```
#pragma HLS INTERFACE ap_ovld port=threshold
```

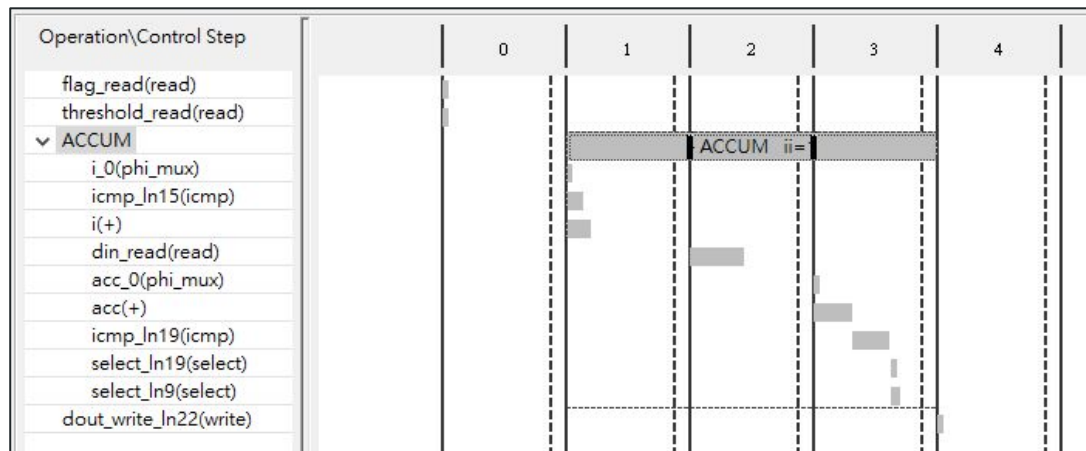
```
#pragma HLS INTERFACE ap_ovld port=dout
```

```
#pragma HLS PIPELINE II=1
```

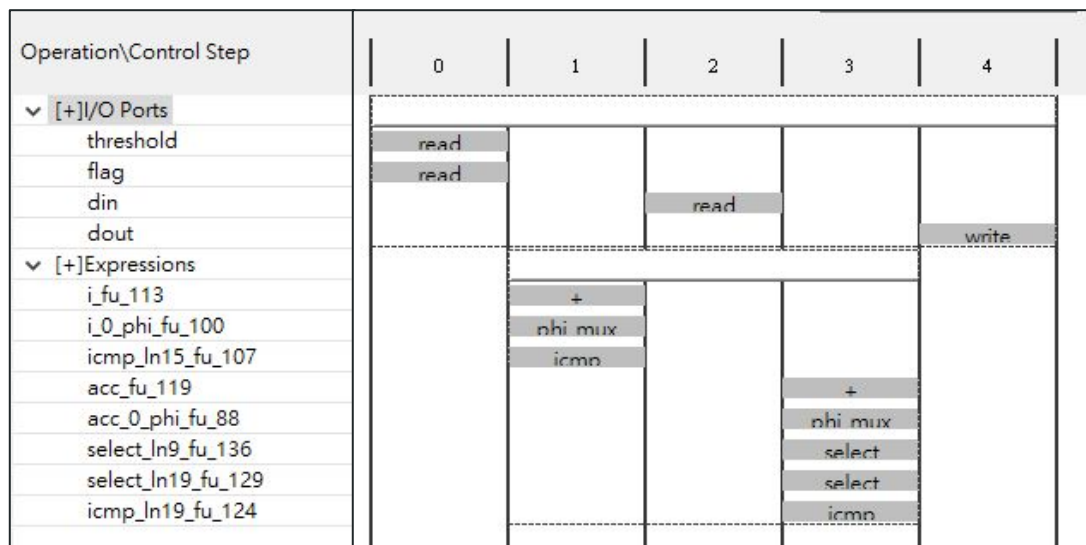
threshold, data mapped to request for data interface

Results

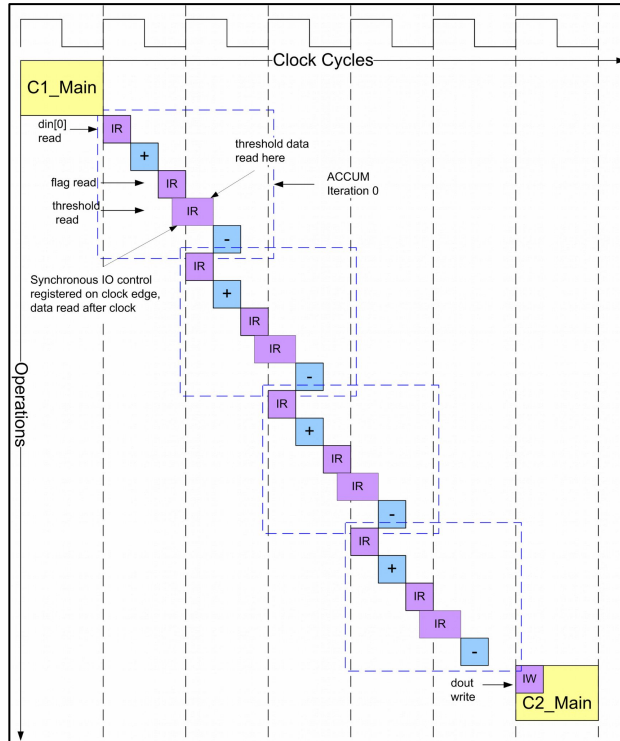
Schedule



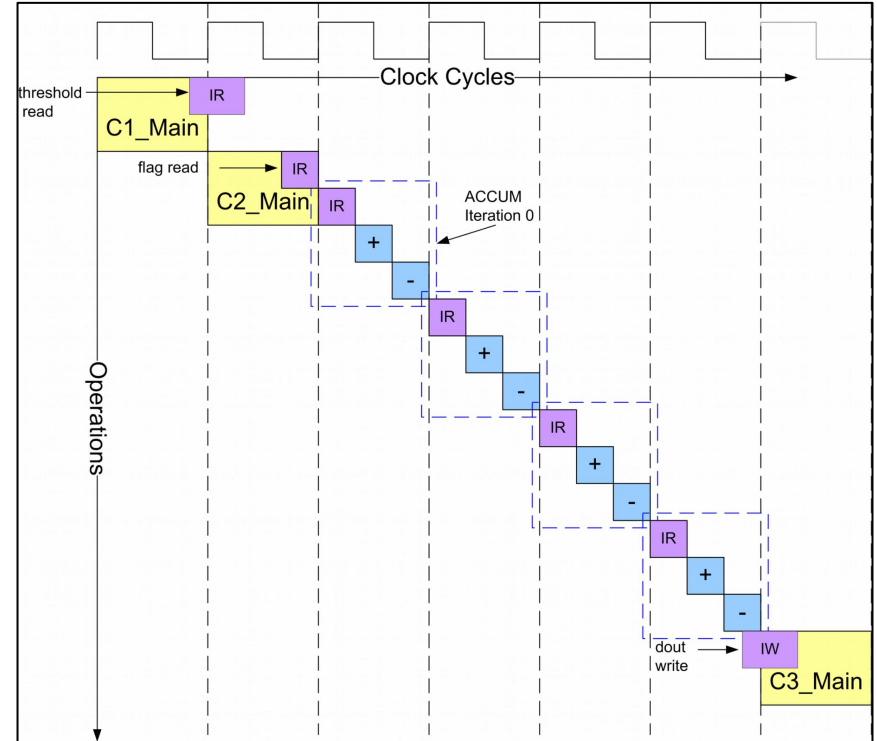
Resource



Expected results



Ref: [HLS Bluebook p.81 Illustration 66](#)



Ref: [HLS Bluebook p.81 Illustration 69](#)

Directives for Cond Pass by Ref. (ready/ack behavior)

```
#pragma HLS INTERFACE ap_fifo port=din
```

```
#pragma HLS INTERFACE ap_none port=flag
```

```
#pragma HLS INTERFACE ap_fifo port=threshold
```

```
#pragma HLS INTERFACE ap_fifo port=dout
```

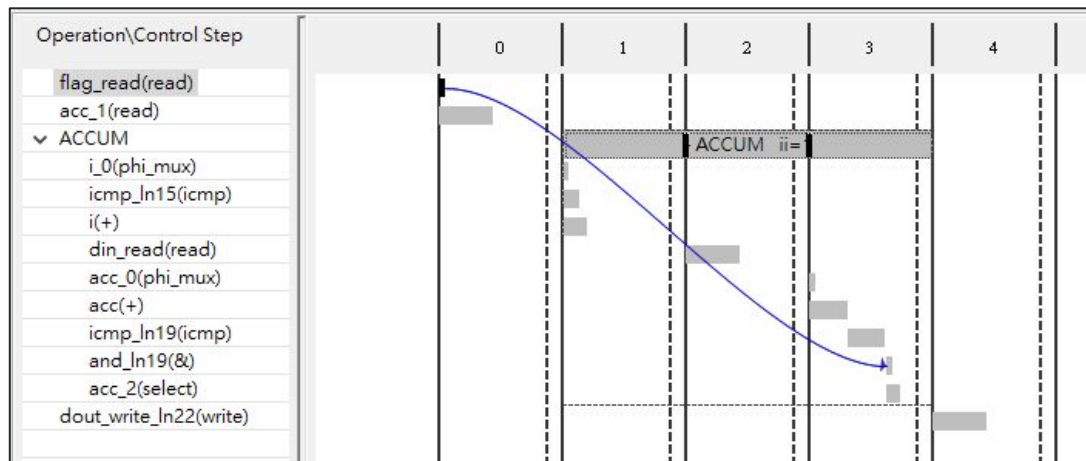
```
#pragma HLS PIPELINE II=1
```

threshold mapped to request-grant interface

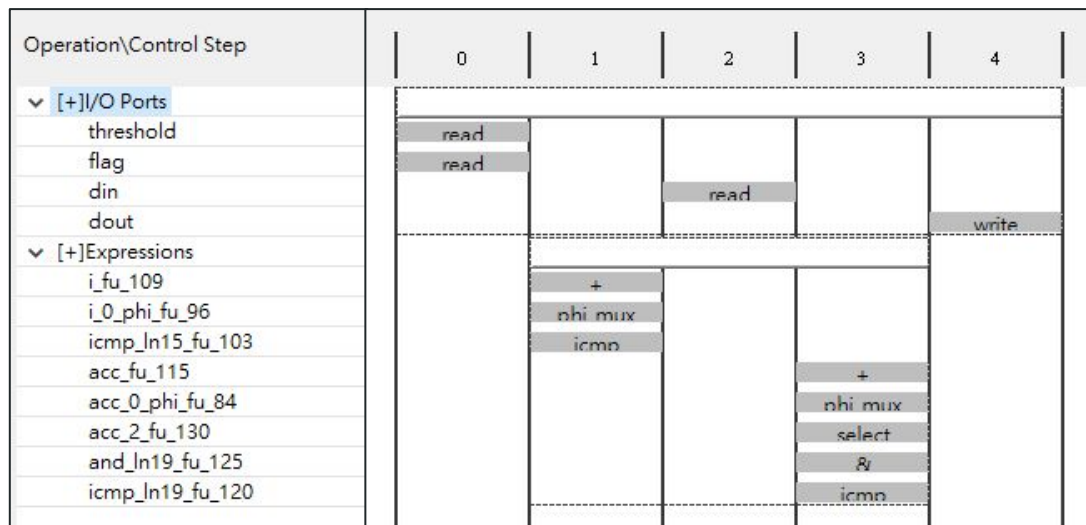
dout mapped to request for data interface

Results

Schedule



Resource



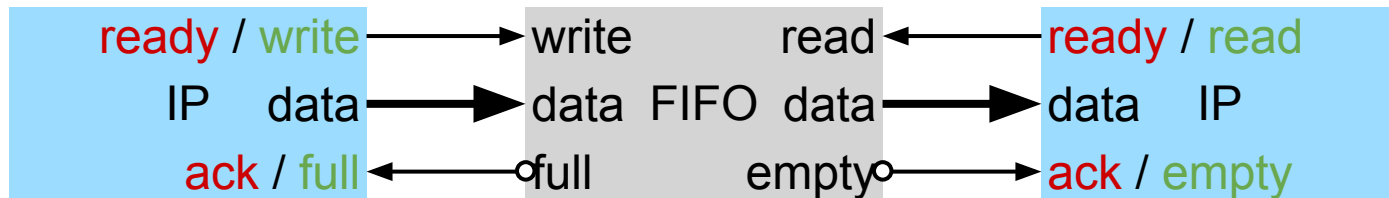
Terminology: “ack” in Catapult C is not “ack” in ap_ack

Catapult C (bluebook)

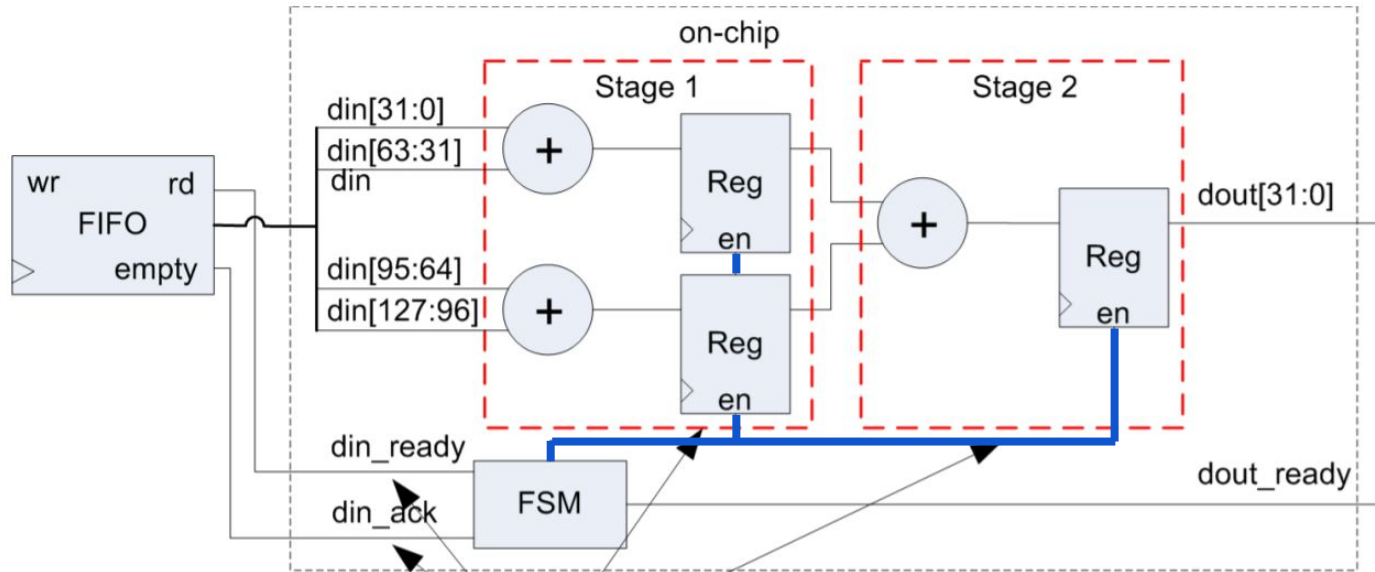
- Input port
 - **ready**(-for-data)
 - **ack**nowledge
- Output port
 - **ready**(-to-send)
 - **ack**nowledge

Vivado HLS

- Input port (FIFO)
 - **read**
 - (not) **empty**
- Output port (FIFO)
 - **write**
 - (not) **full**



5.2.4. Stalling the Pipeline



Stage 1 ready to read data while the FIFO is empty will stall both stage 1 and stage 2, leaving the output stuck in Stage 2

Enables of registers are connected to the same signal

Illustration 76: Hardware of Pipelined Main Loop with Conditional Wait IO


Example 46 Stalling the Pipeline with Conditional IO

```
void accumulate4(int din[4], int &dout){
```

ready-ack

ready

Main loop pipelined with II=1



ap_fifo

```
    ACCUM: for (int i=0; i<4; i++){
```

loop fully unrolled

```
        acc += din[i];
```

```
    }
```

```
    dout = acc;
```

```
}
```

Vivado HLS does not have “ready”-only interface, so the “ready” interfaces are replaced by ready-ack, which is ap_fifo in Vivado HLS

Directives

```
#pragma HLS PIPELINE II=1 [enable_flush]
```

```
#pragma HLS UNROLL
```

```
#pragma HLS INTERFACE ap_fifo port=dout
```

```
#pragma HLS INTERFACE ap_fifo port=din
```

```
#pragma HLS ARRAY_RESHAPE variable=din complete dim=1
```

ARRAY_RESHAPE:

```
input [31:0] din; --> input [127:0] din;
```

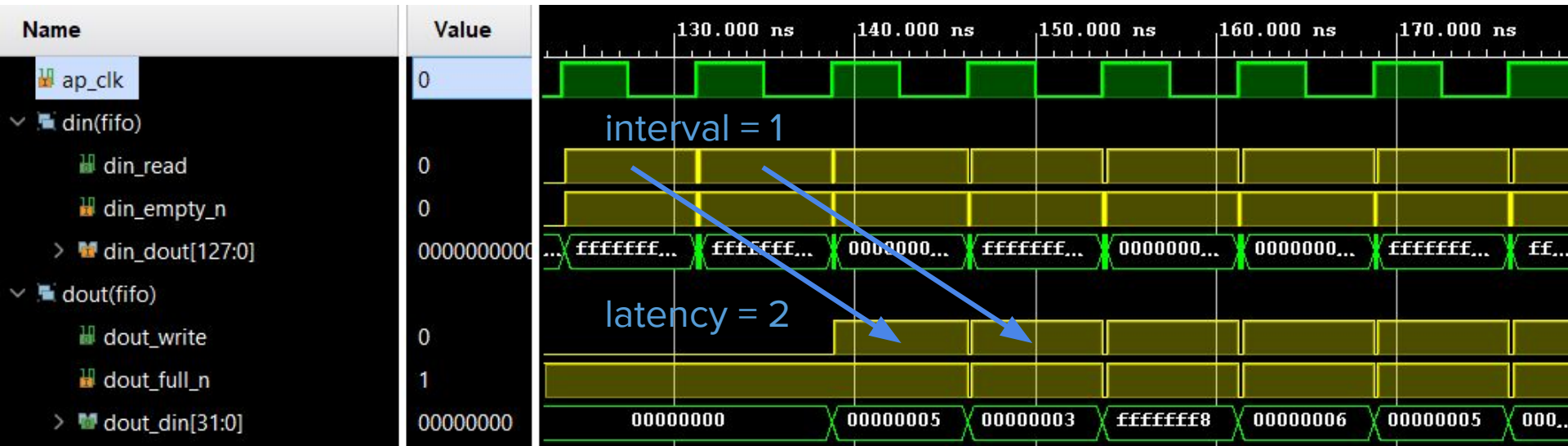
Test bench input

- 7 sets of random integers in $[-9, 9]$

```
int data[] = {  
    2,  5,  3, -5,  
    1,  6,  5, -9,  
   -7,  6, -9,  2,  
    6,  7, -6, -1,  
   -6,  2,  1,  8,  
   -8,  0,  5,  5,  
    4, -9,  5, -7  
};
```

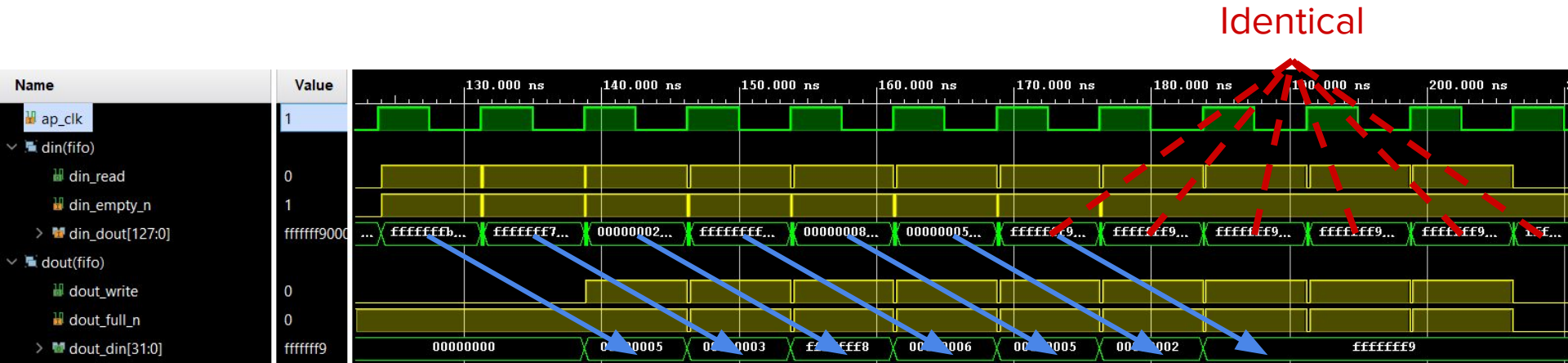

Unsolved problem: control the FIFO in cosim

- Example 46 wants to show that when the input FIFO is empty, the pipeline stalls and doesn't flush
- We don't know how to make the FIFO empty in cosim

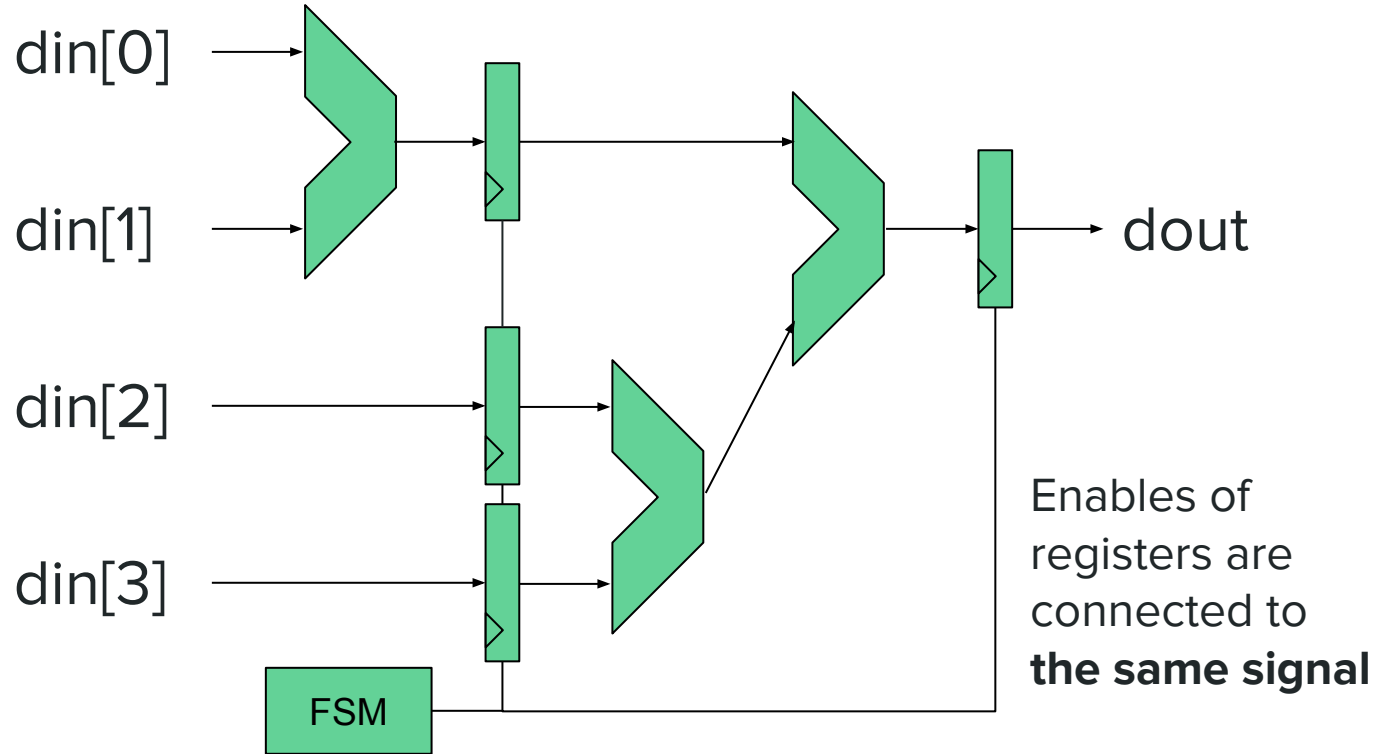


Jiin's advice: feed limited number of inputs and stop

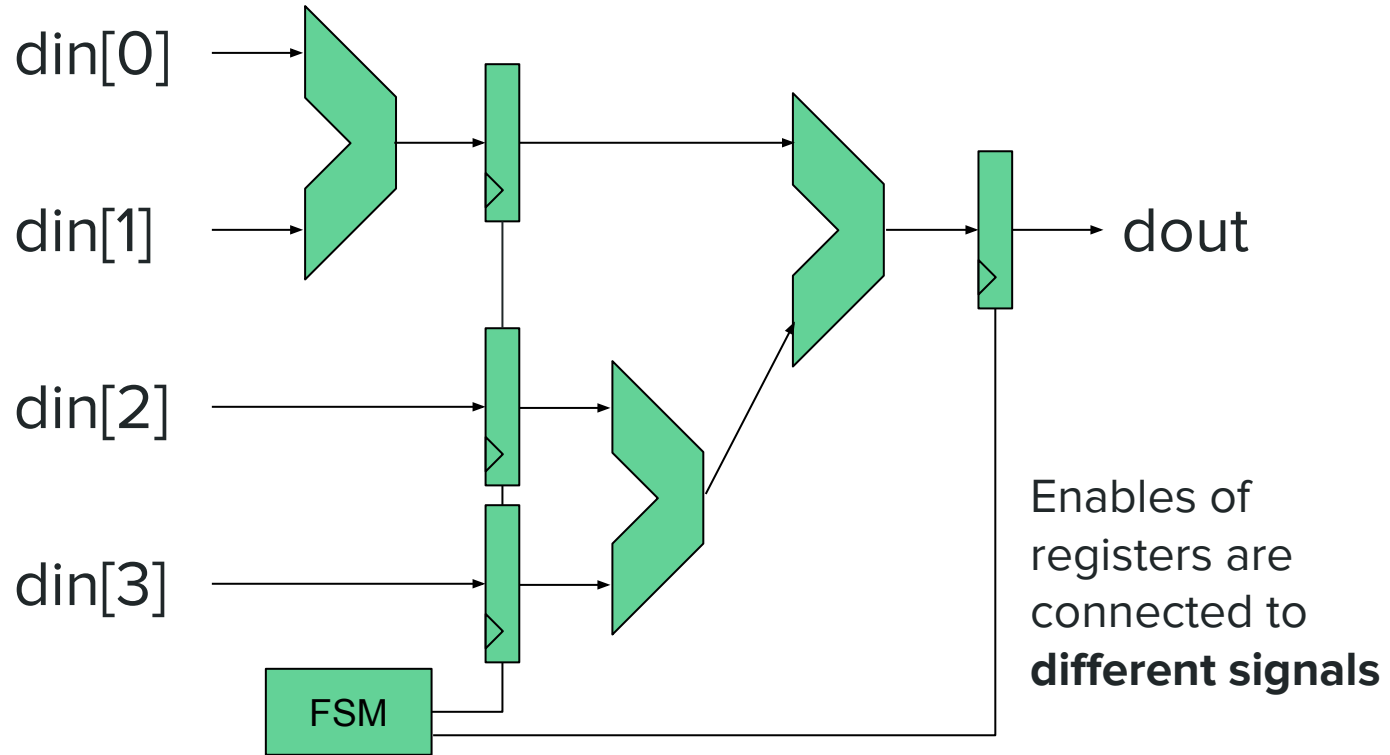
- Same results with or without **enable_flush**
- 4 more reads are performed after the necessary 7 reads
- `din_empty_n` is always high
- The generated Verilog codes are different



Synthesized Verilog code without `enable_flush`



Synthesized Verilog code with `enable_flush`



5.2.5. Manually Flushing the Pipeline

- Problem: pipelining the main loop (without `enable_flush`)
 + using handshaking IO
 = prevent the pipeline from flushing
- Solution: add an explicit ack signal

Example46:

```
void accumulate4(int din[4], int &dout);
```

Example47:

```
void accumulate (int din[4], int &dout, bool &ack);
```

Example 47 Manually Flushing the Pipeline

```
void accumulate(int din[4], int &dout, bool &ack){  
    int acc = 0;    ap_fifo    ap_fifo    ap_none  
    if (ack){  
        ACCUM: for (int i=0; i<4; i++){ loop fully unrolled  
            acc += din[i];  
        }  
        dout = acc;  
    }  
}
```

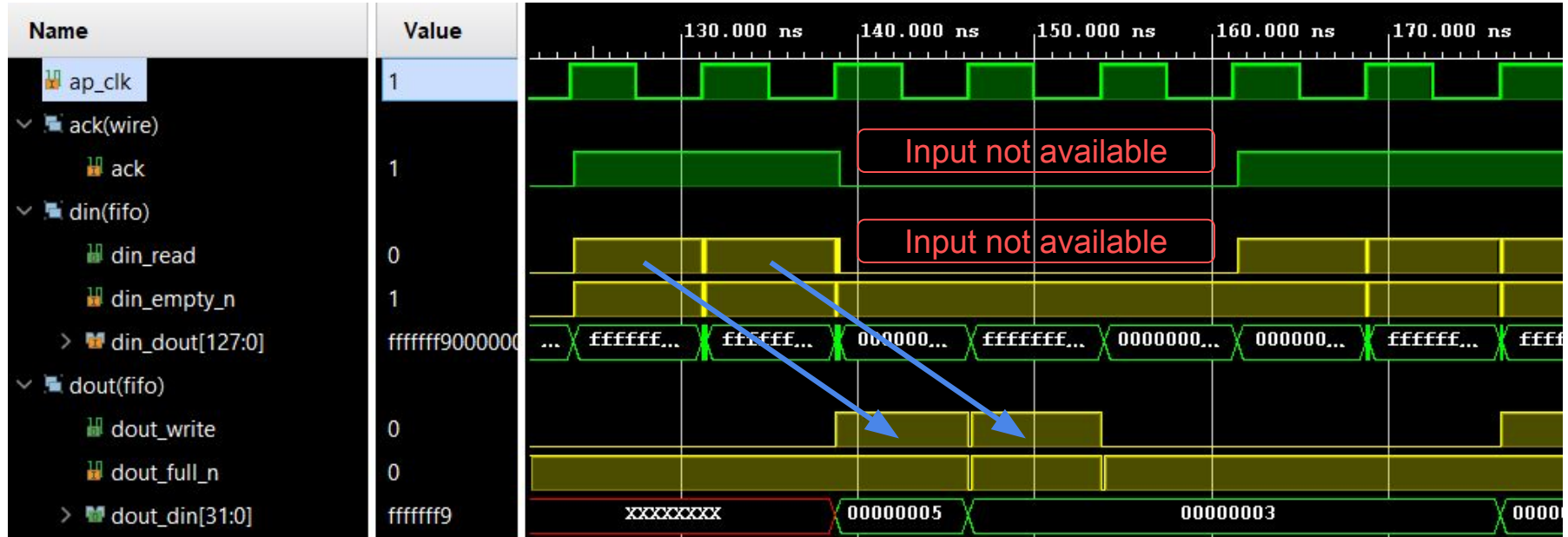
Main loop
pipelined
with II=1

Test bench input

```
bool ack_array[] = {  
    true,  
    true,  
    false,  
    false,  
    false,  
    true,  
    true  
};
```

```
int data[] = {  
    2,  5,  3, -5,  
    1,  6,  5, -9,  
   -7,  6, -9,  2,  
    6,  7, -6, -1,  
   -6,  2,  1,  8,  
   -8,  0,  5,  5,  
    4, -9,  5, -7  
};
```

Pipeline is allowed to flush when input is not available



5.2.6. Writing IO for Throughput

- Mixing conditional logic with IO
 - Data dependency between iterations
 - Hard to pipeline the whole function
- Separating IO from core logic
 - Dependency remains in core logic, no longer affects IO
 - Possible to merge IO => possible to pipeline the whole function

```
for (int i=0; i<4; i++)  
    if (flag[i])  
        acc += din[i];
```

```
for (int i=0; i<4; i++)  
    IO din_int[i] = din[i];  
for (int i=0; i<4; i++)  
    if (flag[i])  
Core    acc += din_int[i];
```

Mixing core logic with IO

- Each read to `din[i]` will occur only if `flag[i]` is set, so they can't be merged
- In the worst case, a read is issued in every clock
- Pipelining prohibited!

```
for (int i=0; i<4; i++)  
    if (flag[i])  
        acc += din[i];
```

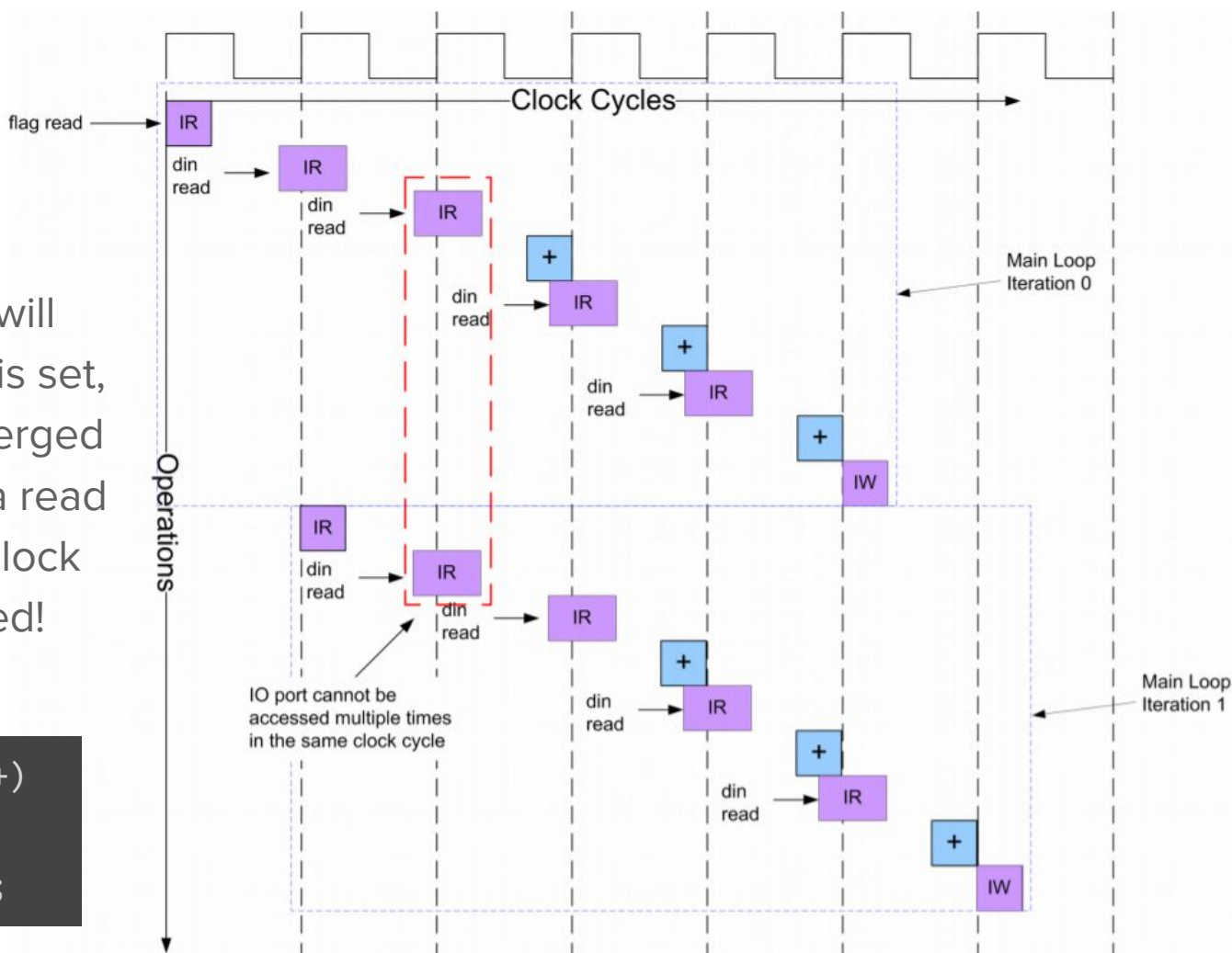


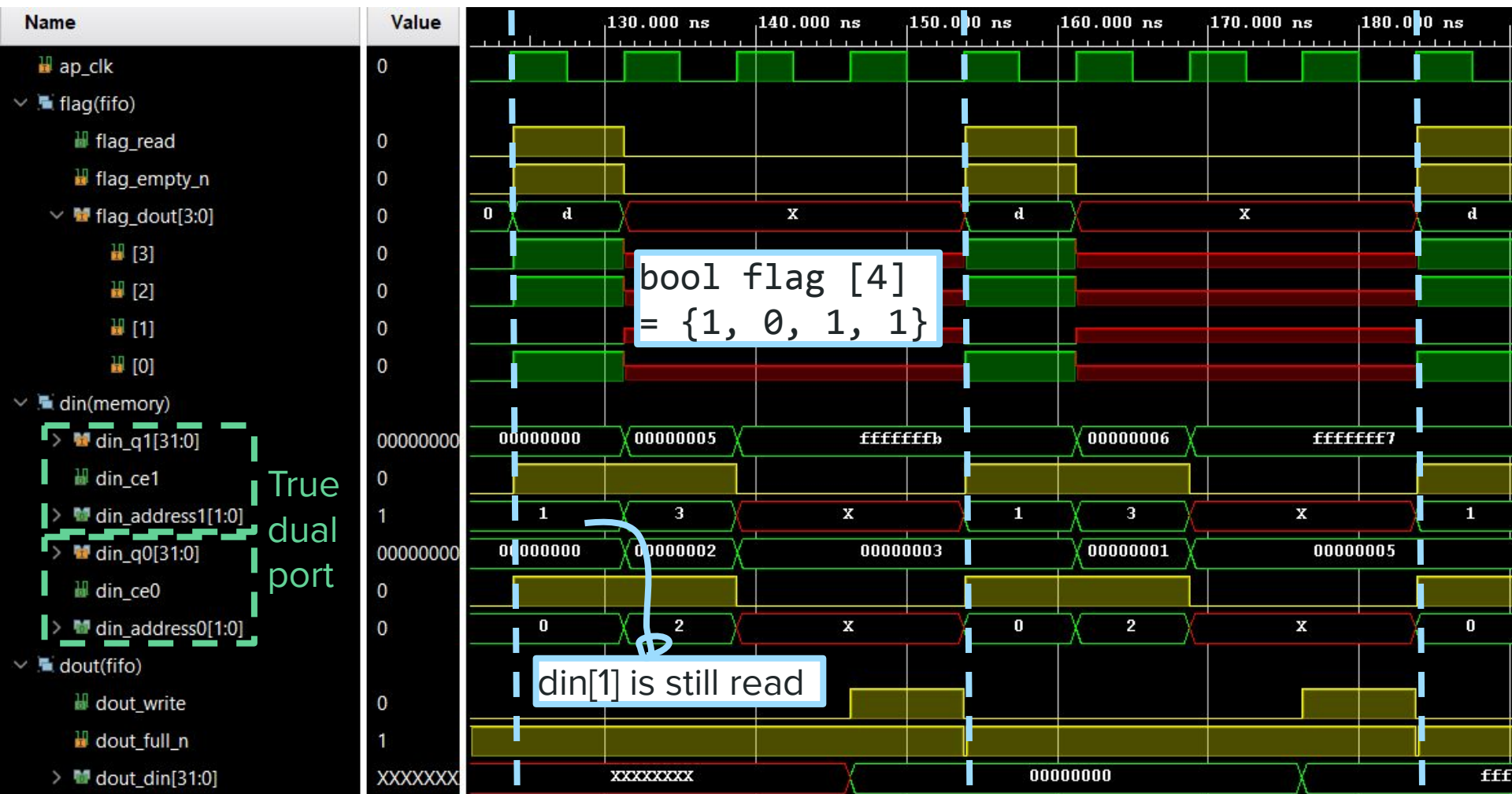
Illustration 79: Design that Can't be Pipelined Due to Unmerged IO

Example 48 IO Throughput Limiting Design

```
void accumulate (int din[4], int &dout, bool flag[4]) {  
    int acc=0; ap_memory ap_fifo ap_fifo  
    ACCUM: for (int i=0; i<4; i++) { loop fully unrolled  
        if (flag[i])  
            acc += din[i];  
    }  
    dout = acc;  
}
```

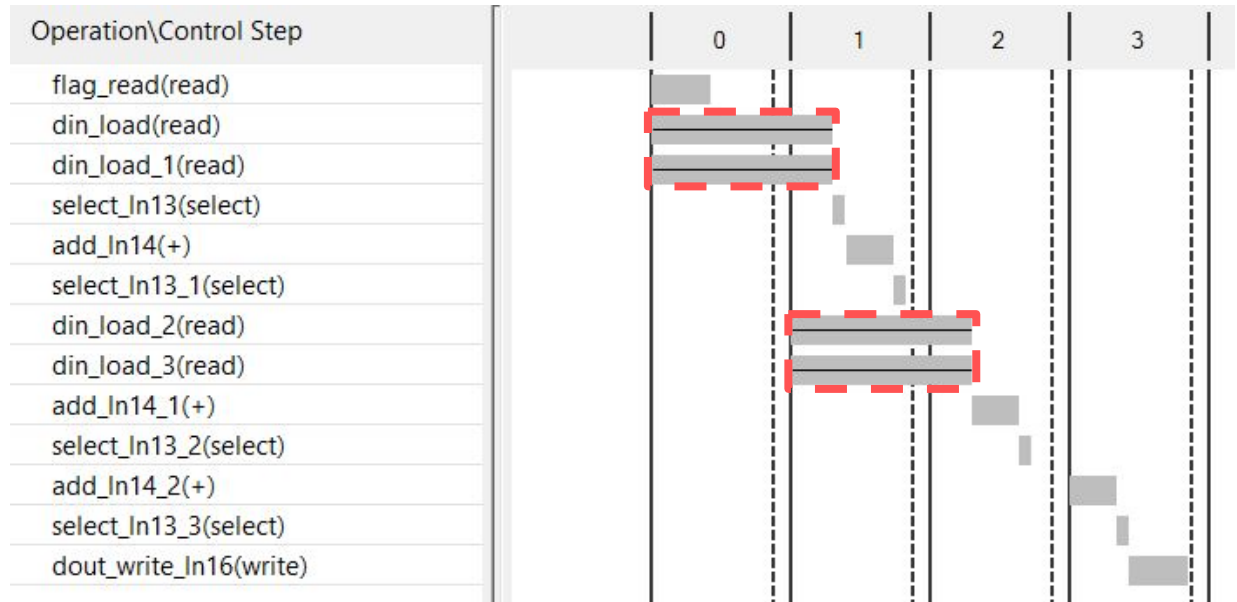
Not sequential access
=> Can't use ap_fifo

Latency = 3
Interval = 3



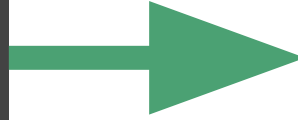
Throughput not limited

- Vivado HLS aggressively reads all elements of din regardless of the conditions



Example 49 Manual Unrolling of IO Throughput Limiting Design

```
// Example 48
for (int i=0; i<4; i++) {
    if (flag[i])
        acc += din[i];
}
```



```
if (flag[0])
    acc += din[0];
if (flag[1])    dependency
    acc += din[1];
if (flag[2])    dependency
    acc += din[2];
if (flag[3])    dependency
    acc += din[3];
```

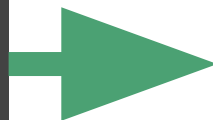
Hand-drawn green arrows on the right side of the unrolled code indicate dependencies between the 'if' statements and their corresponding 'acc += din[i];' statements. The arrows show a sequential flow from the first 'if' to the first assignment, then to the second 'if', and so on, illustrating the data dependencies in the unrolled version.

- Identical waveform, same latency and interval
- Identical resource usage

Example 50 Making IO Read Unconditional

```
// Example 48
```

```
ACCUM: for (int i=0; i<4; i++) {  
    if (flag[i])  
        acc += din[i];  
}
```

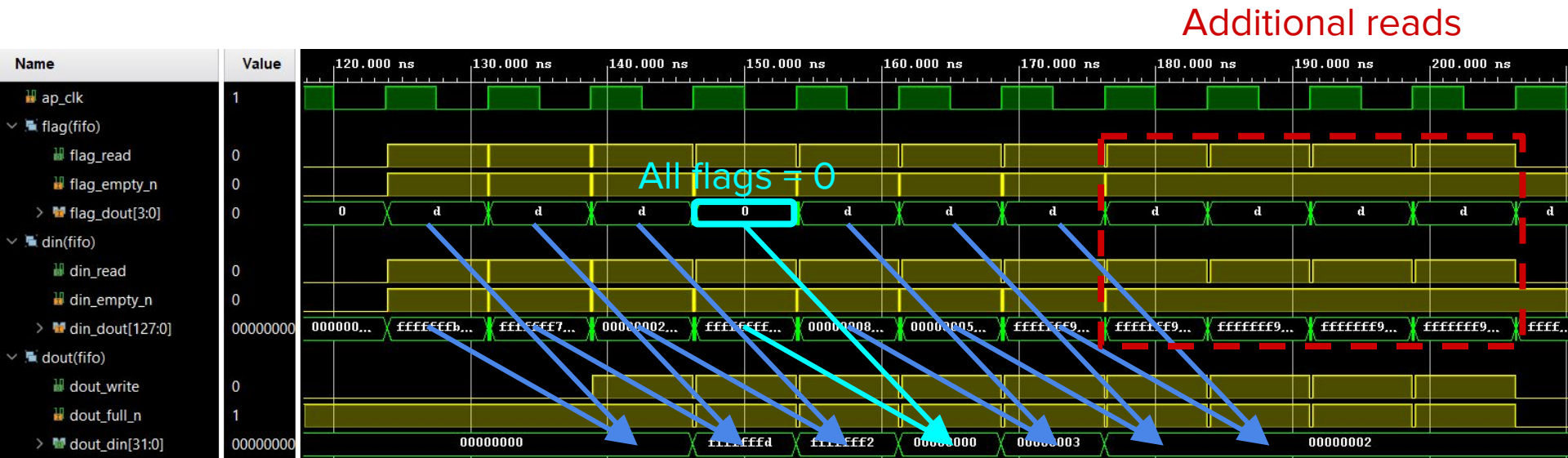


```
int din_int[4];  
DIN: for (int i=0; i<4; i++)  
    din_int[i] = din[i];  
  
ACCUM: for (int i=0; i<4; i++){  
    if (flag[i])  
        acc += din_int[i];  
}
```

- Merge of din IO allowed => Main loop pipelined with ll=1
- Inner loops fully unrolled

Example 50: waveform

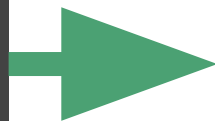
- Latency = 2, Interval = 1 (3 times throughput)
- In the 4th input, all flags = 0, so there's no need for reading din



Example 51 Simplifying Conditional IO to Help Merging

```
// Example 50
```

```
DIN: for (int i=0; i<4; i++)  
    din_int[i] = din[i];
```



```
bool flag_int;  
FLAG: for (int i=0; i<4; i++)  
    flag_int |= flag[i];  
  
DIN: for (int i=0; i<4; i++)  
    if (flag_int)  
        din_int[i] = din[i];
```

- Same constraints as Example 50
- If all flags = 0, do not read din

Example 51: waveform

- `din` is still read when all flags = 0
- In Example 51, Vivado HLS can't detect unnecessary reads to `din`
- In Example 47, Vivado HLS skips the whole function if `ack` is low

```
// Example 51
void accumulate(..., bool flag[4]){
    int acc=0;
    int din_int[4];
    bool flag_int;
    for (int i=0; i<4; i++)
        flag_int |= flag[i];
    for (int i=0; i<4; i++)
        if (flag_int)
            din_int[i] = din[i];
    ...
}
```

```
// Example 47
void accumulate(..., bool &ack){
    int acc = 0;
    if (ack){
        ...
    }
}
```

Resource usage comparison

	Example 48, 49	Example 50, 51
Pipelined	No	Yes
Latency (cycles)	3	2
Interval (cycles)	3	1
Expression (LUT)	247	251
Register (FF)	104	199
din interface	ap_memory	ap_fifo
Multiplexer (LUT)	75	27

Memories

5.3.1 Automatic Mapping of Arrays to Memories

```
void accumulate4(int din[4], int &dout){  
    int acc = 0;  
  
    ACCUM: for (int i=0; i<4; i++){  
        acc += din[i];  
    }  
    dout = acc;  
}
```

single port RAM
interface

ready

Main loop
pipelined
with II=1

Directives

```
set_directive_interface -mode ap_memory "accumulate4" din
```

```
set_directive_interface -mode ap_fifo "accumulate4" dout
```

```
set_directive_resource -core RAM_1P "accumulate4" din
```

```
set_directive_pipeline -II 1 "accumulate4"
```

Hardware of Arrays Mapped to Memories

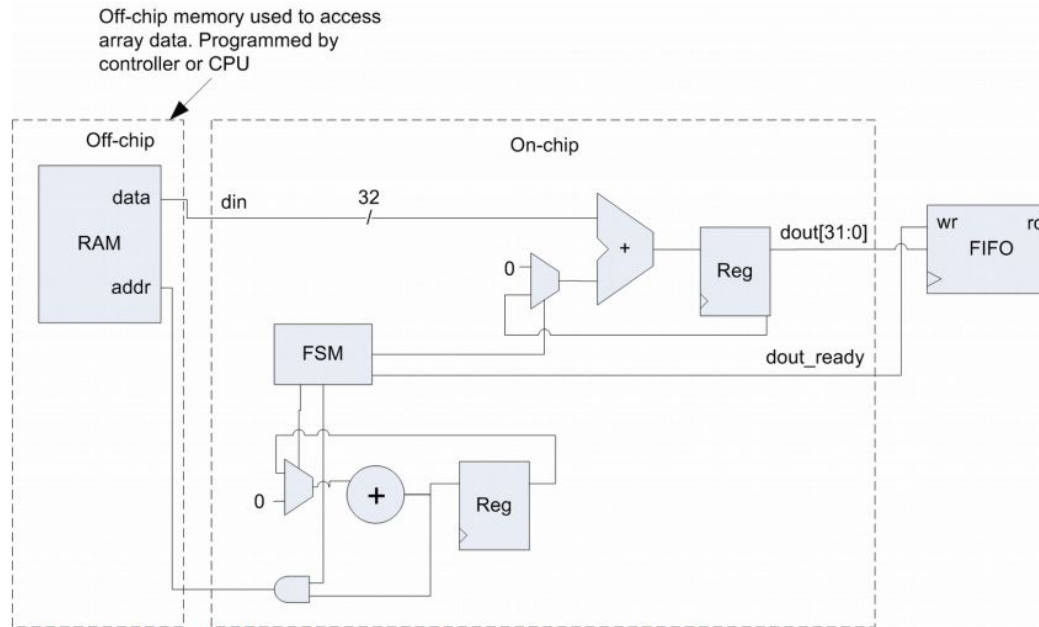


Illustration 83: Hardware of Arrays Mapped to Memories

Only one memory read per iteration of ACCUM, which allows the design to be pipelined with $II=1$.

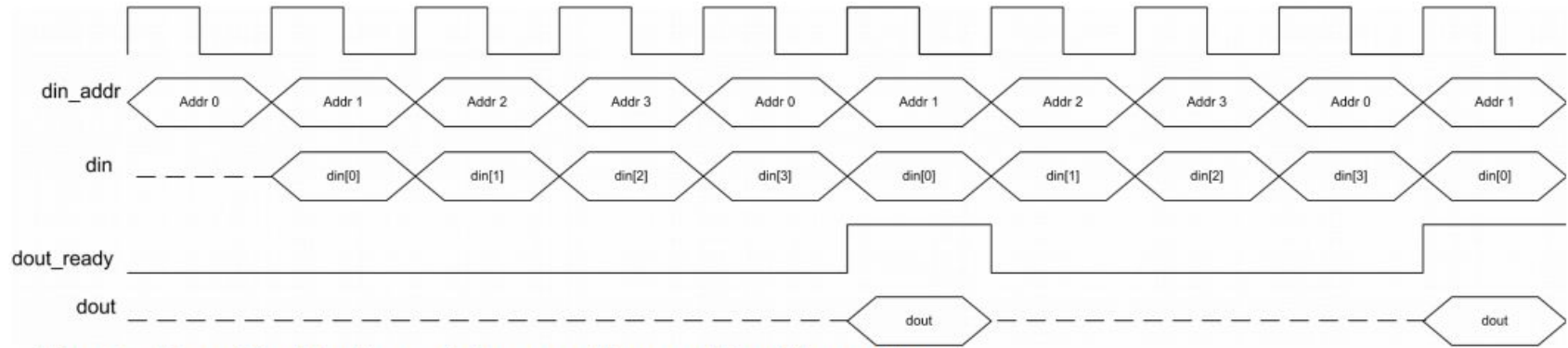
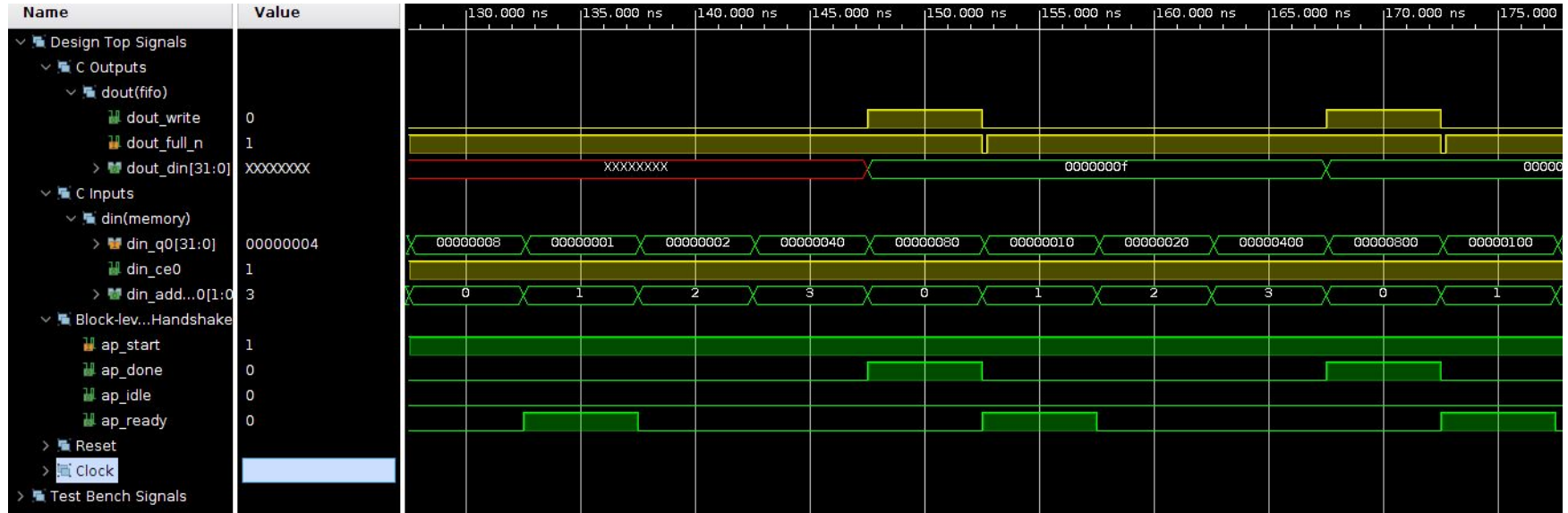


Illustration 82: Timing of Arrays Mapped to Memories

Only one memory read per iteration of ACCUM, which allows the design to be pipelined with $II=1$.



5.3.2 Automatic Memory Merging

```
void accumulate4(int din[4], int &dout){
```

```
    int acc = 0;
```

single port RAM
interface

ready

Main loop
pipelined
with II=1

```
    ACCUM: for (int i=0; i<4; i++){
```

loop unrolled by 2

```
        acc += din[i];
```

```
    }
```

```
    dout = acc;
```

```
}
```

Directives

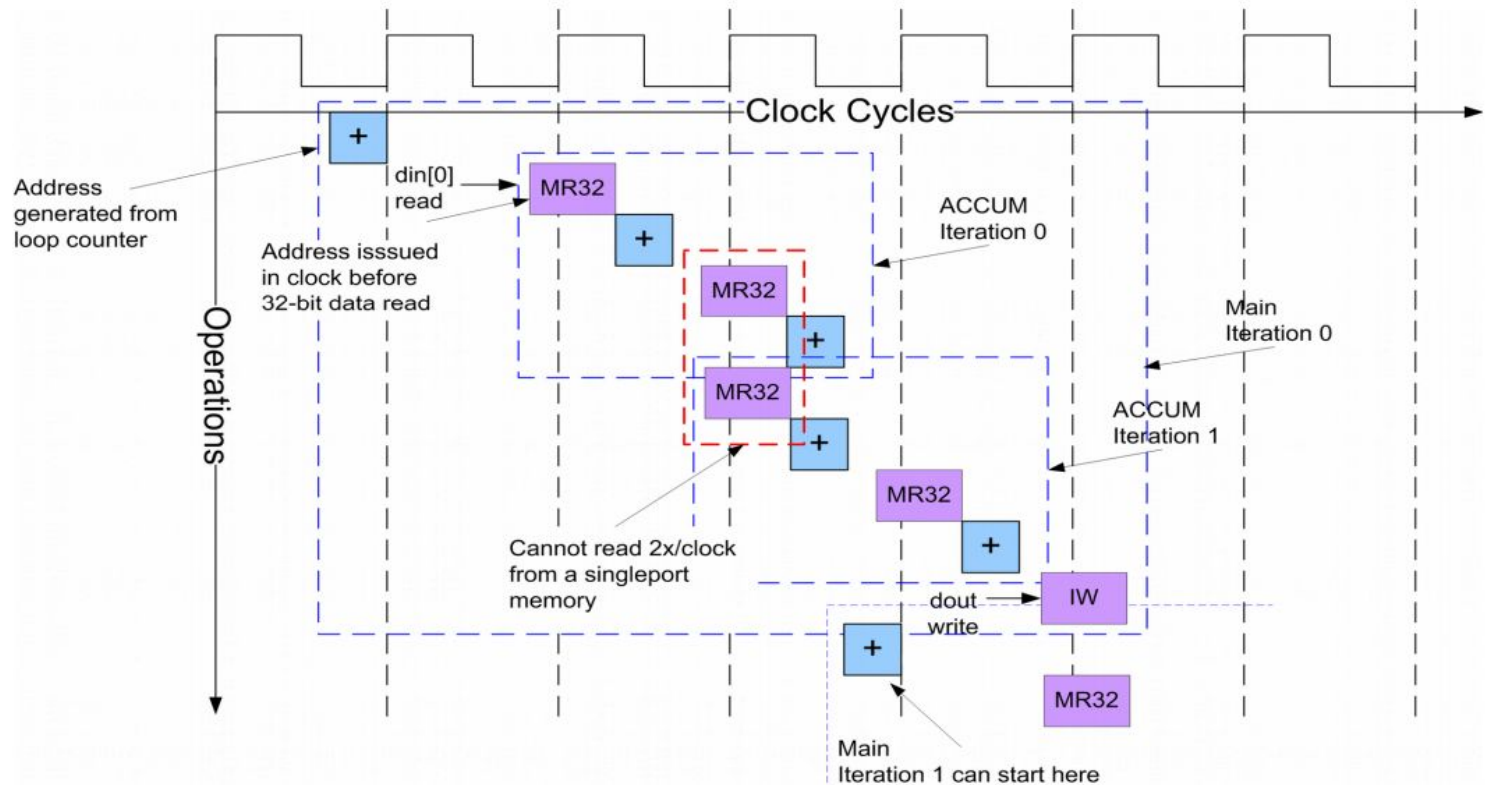
```
set_directive_unroll -factor 2 "accumulate4/ACCUM"
```

```
set_directive_interface -mode ap_memory "accumulate4" din
```

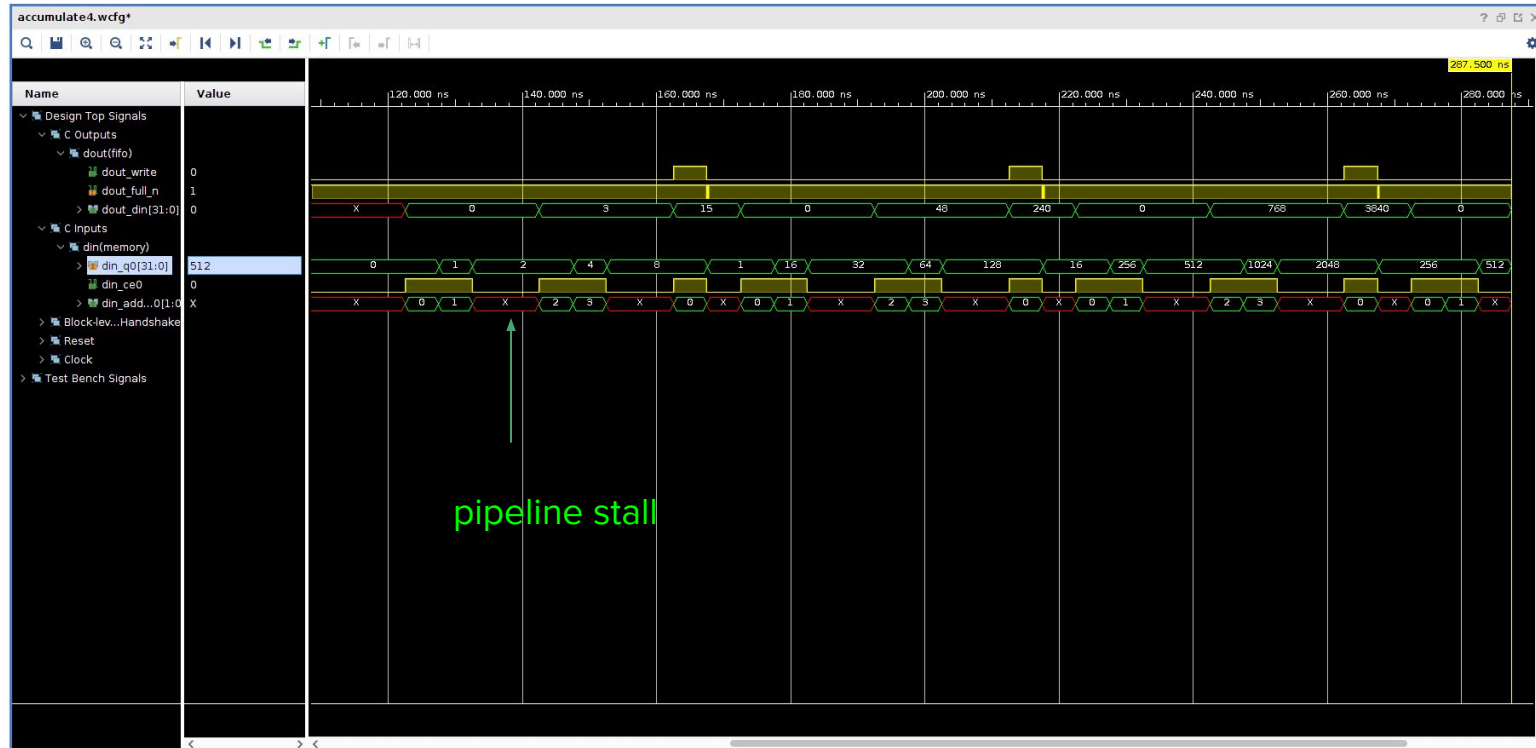
```
set_directive_interface -mode ap_fifo "accumulate4" dout
```

```
set_directive_resource -core RAM_1P "accumulate4" din
```

Failed Schedule for Unmerged Memory Accesses with $II=1$



Waveform



Solution: Widening the word width of the memory interface to 64 bits

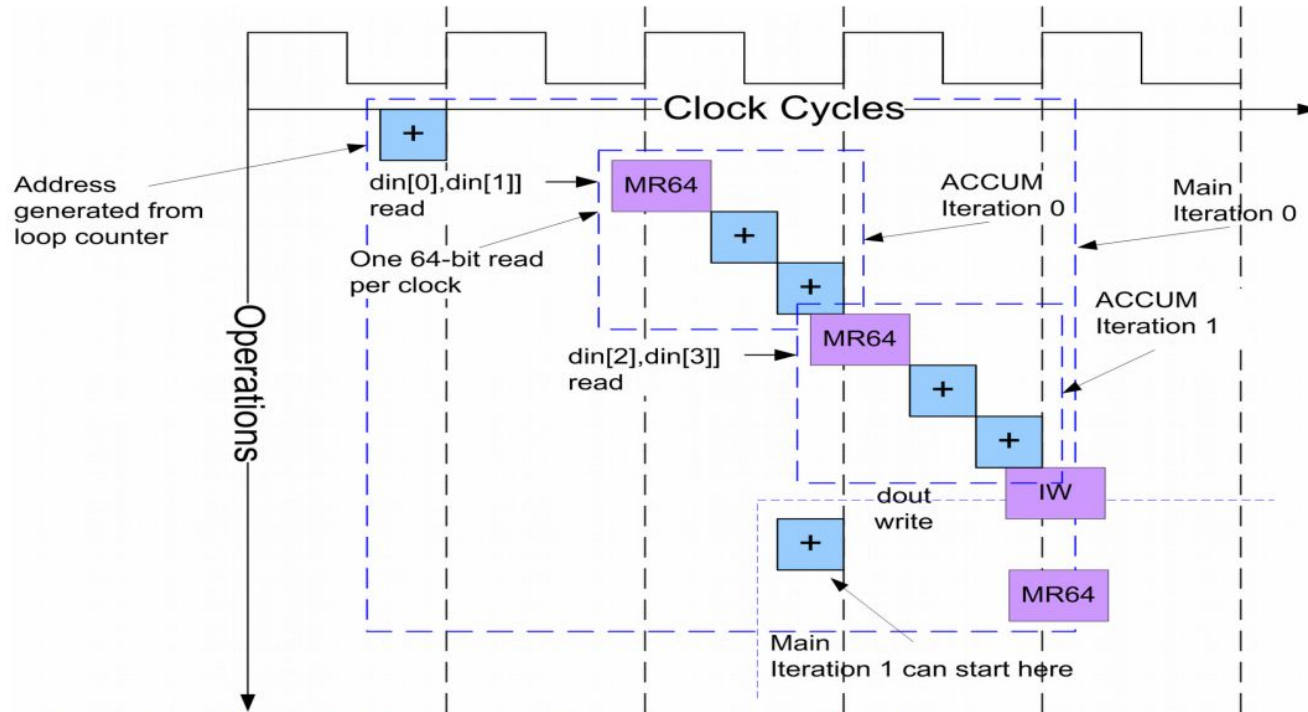


Illustration 86: Schedule for Merged Memory Accesses with $II=1$

Directives

```
set_directive_unroll -factor 2 "accumulate4/ACCUM"
```

```
set_directive_interface -mode ap_memory "accumulate4" din
```

```
set_directive_interface -mode ap_fifo "accumulate4" dout
```

```
set_directive_resource -core RAM_1P "accumulate4" din
```

```
set_directive_pipeline -II 1 "accumulate4"
```

```
set_directive_array_reshape -type cyclic -factor 2 -dim 1  
"accumulate4" din
```

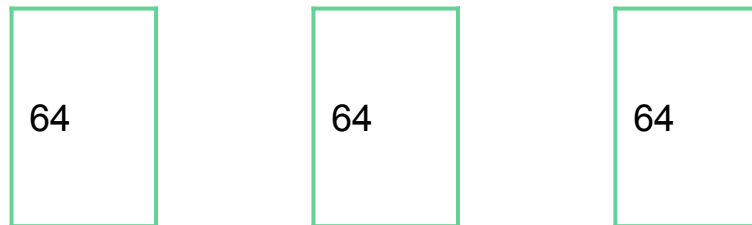
Array reshape

Array reshape = Array partition + Array Map vertical

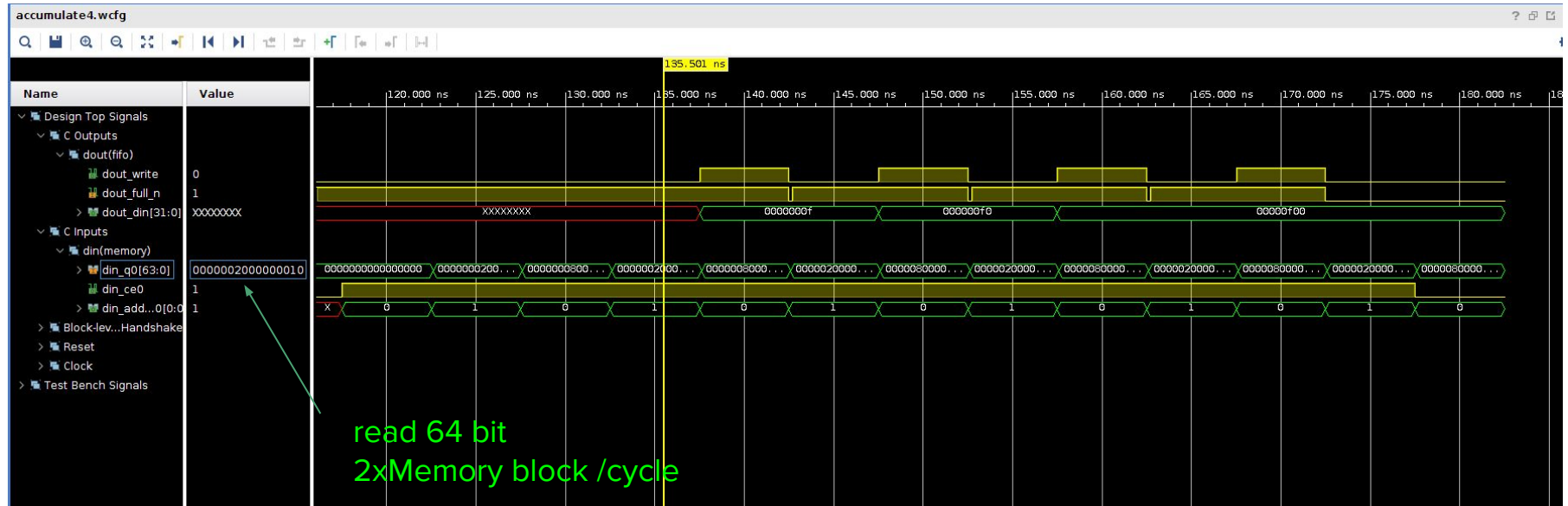
Array partition



Array Map Vertical



Waveform



5.3.3 Designing for Throughput When Using Memories

```
#include "mem_non_mutual_exclusive.h"
```

```
void accumulate (int din[4],  
                 int &dout,  
                 bool &flag0,  
                 bool &flag1){
```

Main loop
pipelined
with ll=1

```
    int acc=0;  
    ACCUM: for (int i=0; i<4; i++){  
        if (flag0)  
            acc += din[i];  
        if (flag1)  
            acc -= din[i]/2;  
    }  
    dout = acc;
```

Non-mutually
Exclusive Memory
Accesses

```
}
```

Directives

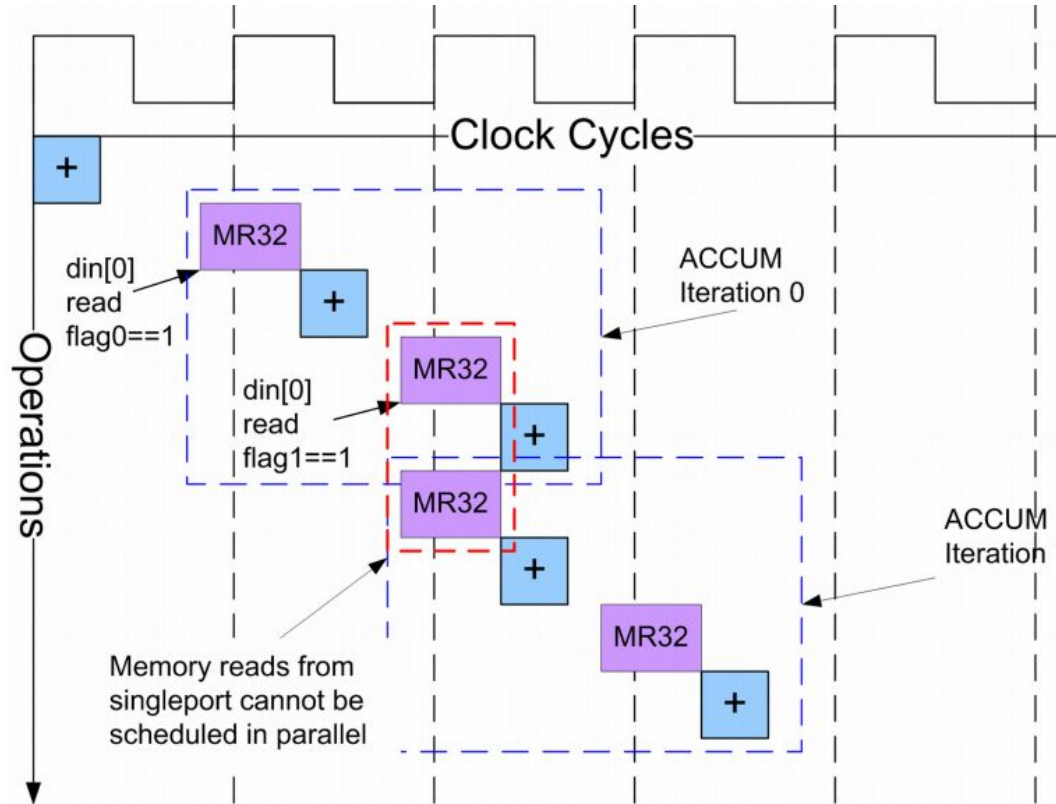
`set_directive_pipeline -ll 1 "accumulate"`

`set_directive_interface -mode ap_memory "accumulate" din`

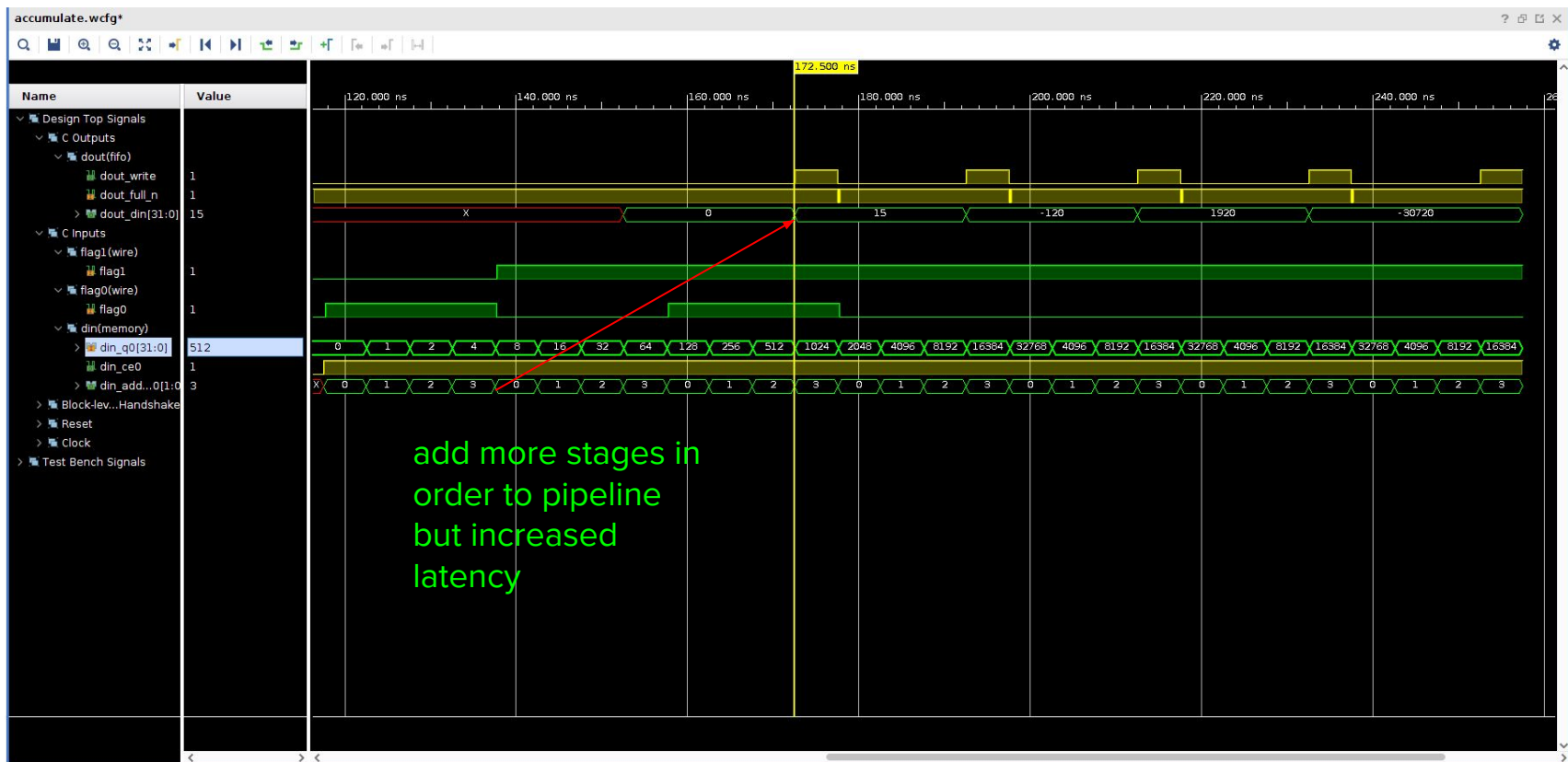
`set_directive_resource -core RAM_1P "accumulate" din`

`set_directive_interface -mode ap_fifo "accumulate" dout`

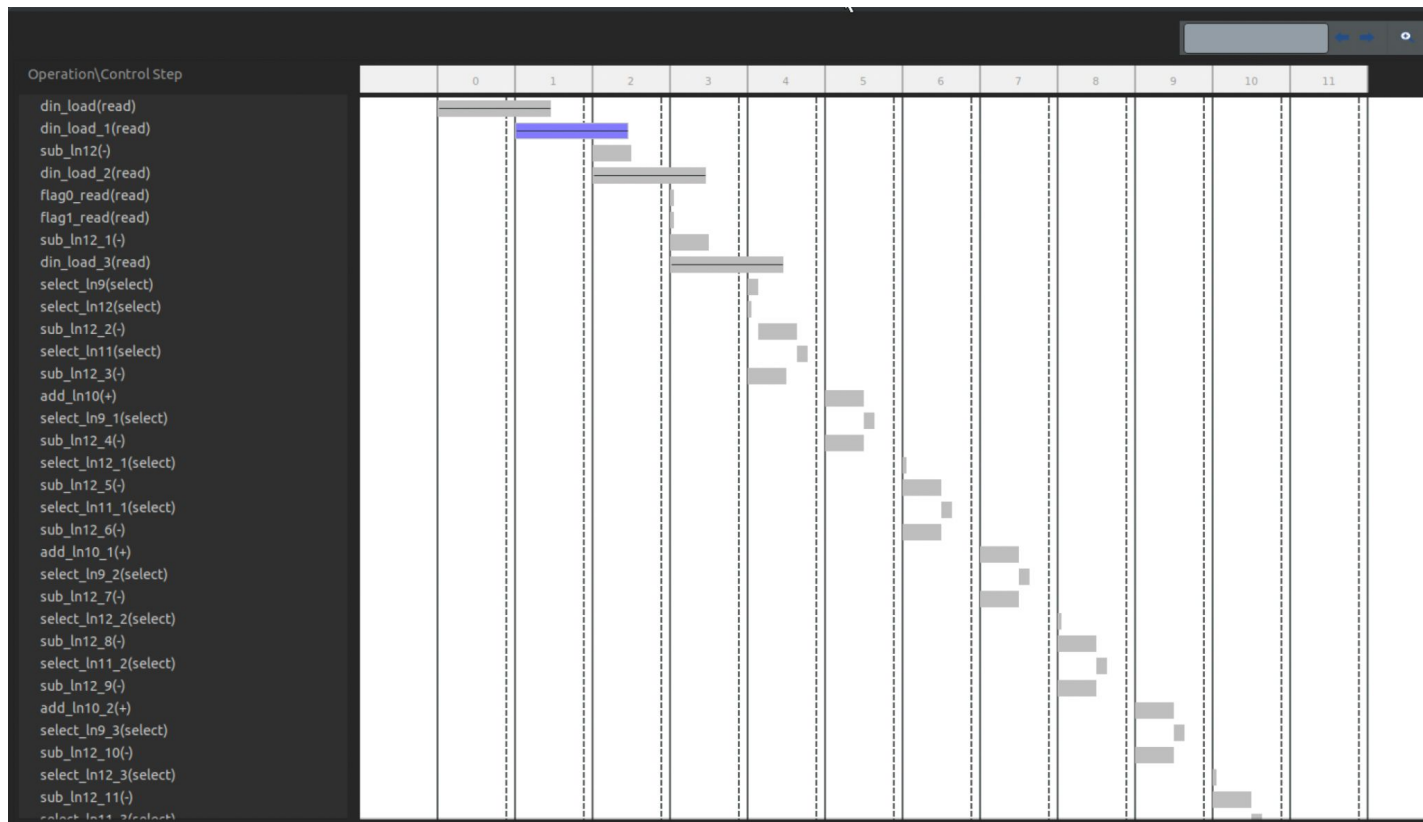
Failed Schedule for Non-mutually Exclusive Memory Accesses



Waveform



Schedule Viewer

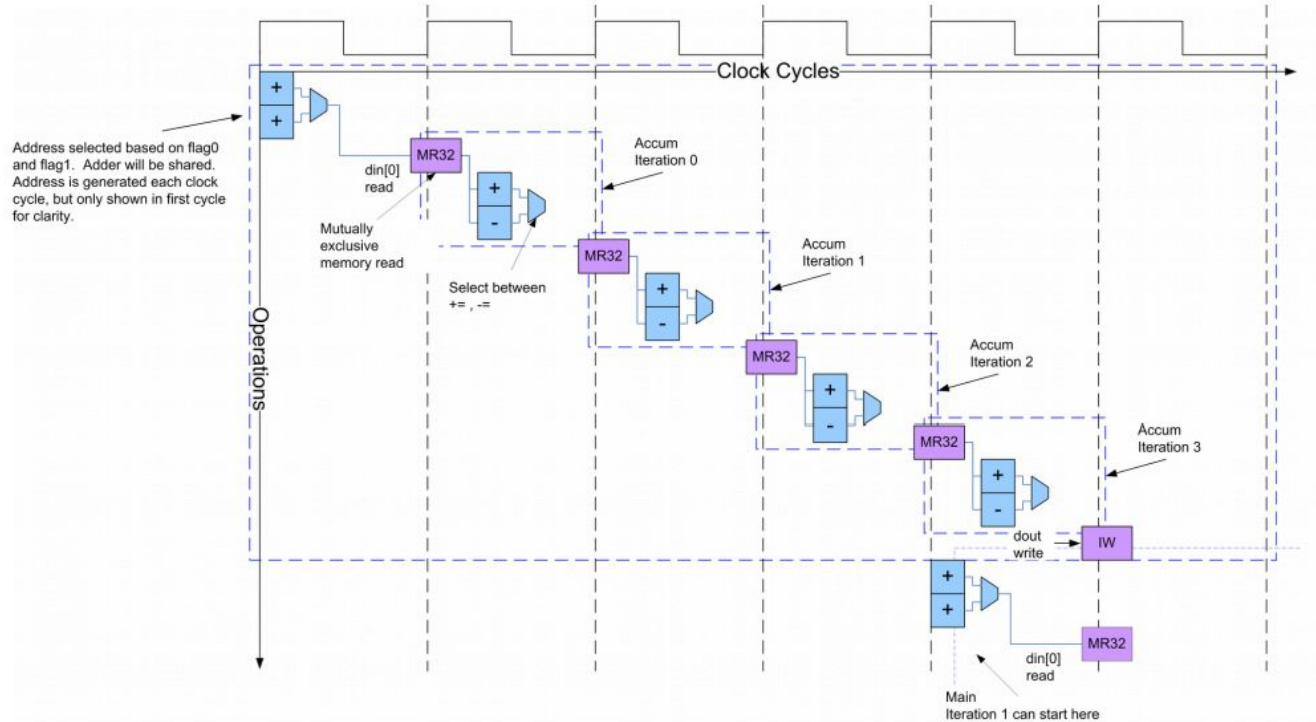


Solution1: Make it Exclusive!

```
#include "mem_mutual_exclusive.h"

void accumulate(int din[4],
               int &dout, bool
               &flag0, bool
               &flag1){
    int acc=0;
    ACCUM: for (int i=0; i<4; i++){
        if (flag0)
            acc += din[i];
        else if (flag1)
            acc -= din[i]/2;
    }
    dout = acc;
}
```

Solution1: Make it Exclusive!



Waveform



Solution2: Manually Merging Non-Mutually Exclusive Memory Accesses

```
#include "mem_non_mutual_exclusive_manual.h"

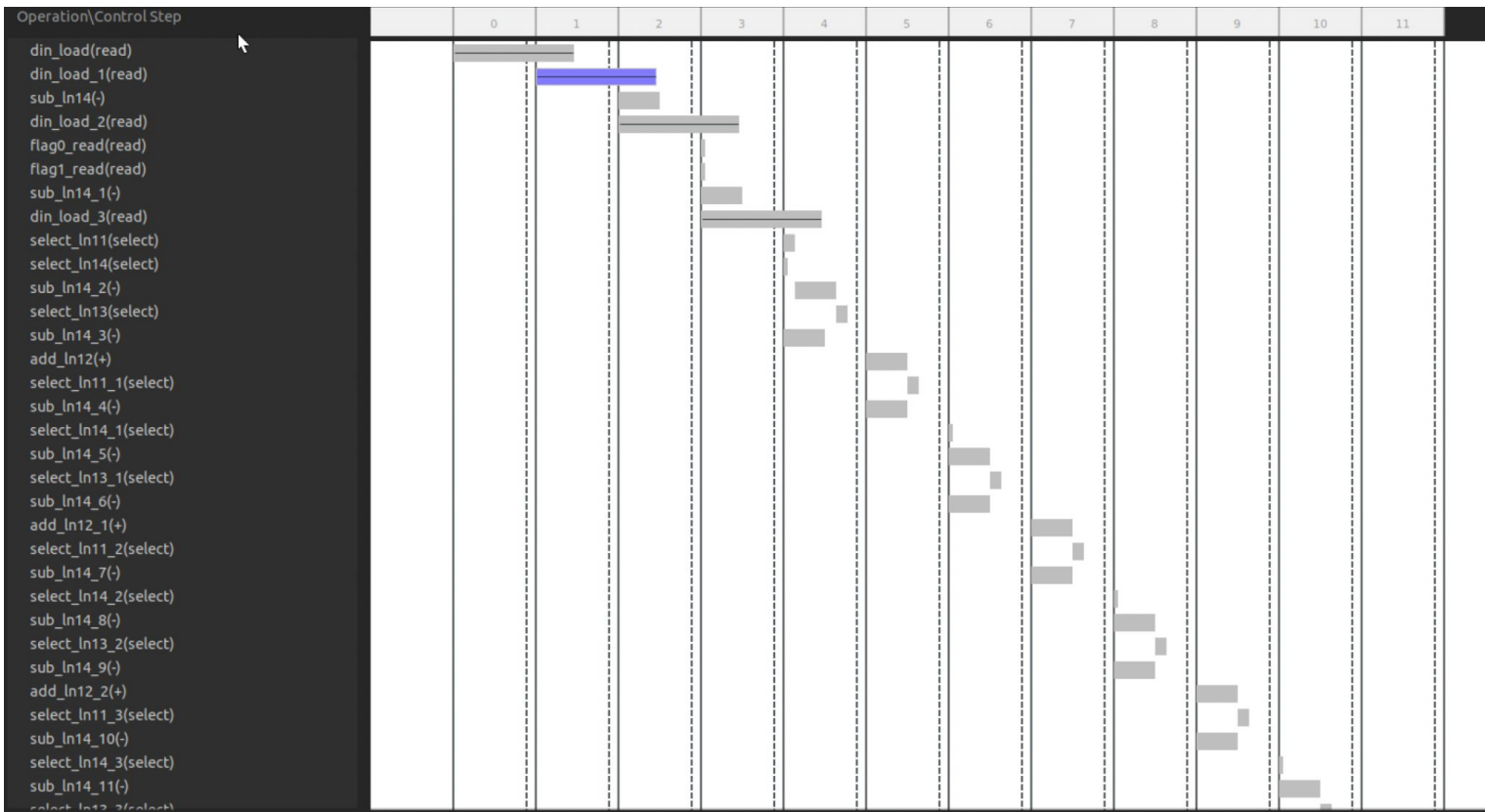
void accumulate (int din[4],
                 int &dout,
                 bool &flag0,
                 bool &flag1){
    int acc=0;
    int tmp;
    ACCUM: for (int i=0; i<4; i++){
        tmp = din[i];
        if (flag0)
            acc += tmp;
        if (flag1)
            acc -= tmp/2;
    }
    dout = acc;
}
```

This technique reduce the number of reads of "din" to once per loop iteration. But can only be used when the original design access the same address for both read of din.

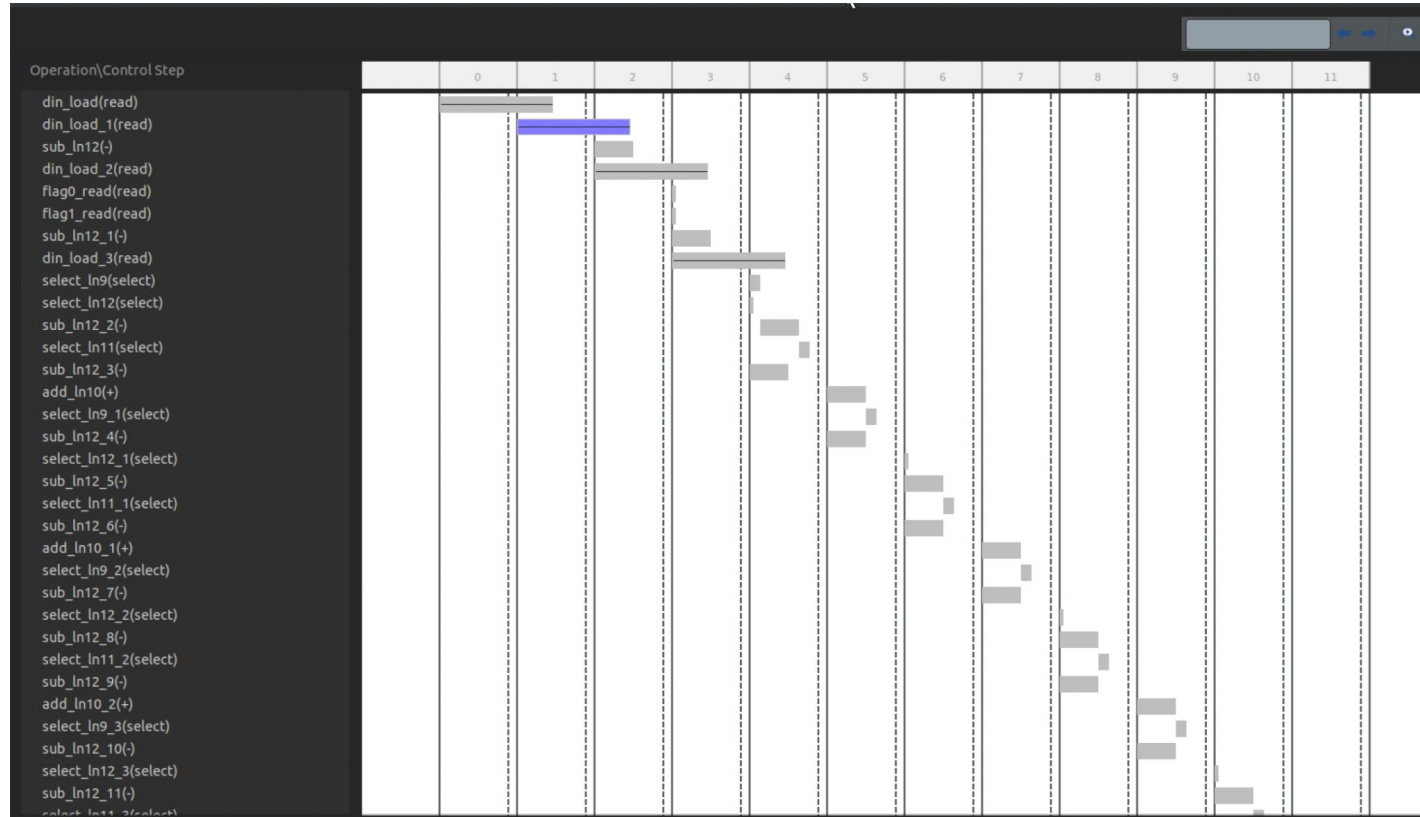
Waveform



Schedule viewer



Compare to compiler automatically optimize



Thank you for listening!
