Memory Access Scheduling and Loop Pipelining

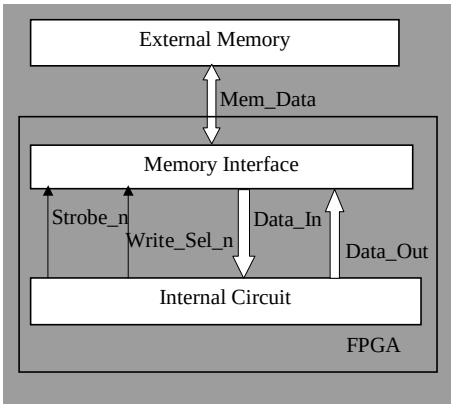
Xuejun Liang* and Jack Jean**
*Jackson State University, Jackson, MS
** Wright State University, Dayton, OH

Outline

- Introduction to the Problem
- Results
- Algorithms
  - Memory Access Control Scheduling
  - Loop Pipelining with Modulo Scheduling
  - Standardizing Memory Access Schedule
- Conclusions

Introduction

• Computation Model with FPGA



• Memory Access Schedule

Clk	1	2	3	4	5	6
Activity	R	R	R	C	C	W

• Control Signal Schedule

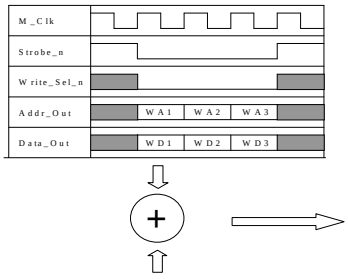
Clk	1	2	3	4	5	6
Strobe_n						
Write_Sel_n						
Addr						

• Memory Access Conflict

Introduction (Cont.)

• Memory Access Control Scheduling

Sample Timing Diagrams



Control Signal Schedule

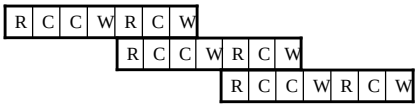
Clk	1	2	3	4	5	6
Strobe_n						
Write_Sel_n						
Addr						

Memory Access Schedule

Clk	1	2	3	4	5	6
Activity	R	R	R	C	C	W

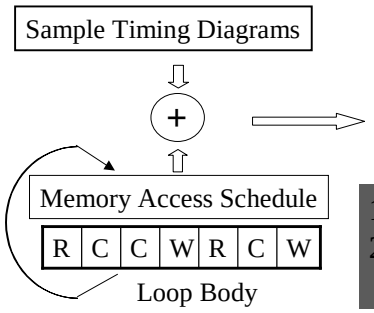
Introduction (Cont.)

- Loop Pipelining with Modulo Schedule



- Initiation Interval (II)
- Minimum II

- Memory Access Control Scheduling with Loop Pipelining



Control Signal Schedule
- Prologue
- Steady State
- Epilogue

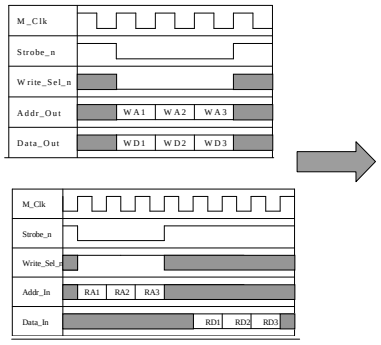
1. What is the minimum II?
2. How to produce the control signal schedule automatically?

Results

1. It is proved that the minimum II is the number of memory accesses by the loop body
2. The minimum II can be achieved by Memory access scheduling
   – Rearrange the ordering of memory accesses
3. Memory access scheduling is performed after the circuit synthesis for the loop computation.
   - Reduce the complexity of the circuit synthesis
   - Support the design portability
4. Algorithms are provided to produce the memory access control schedule automatically.

Memory Access Timing Rules

Sample Timing Diagrams



Rule 1: Default
Strobe_n is 1 (high)
Write_Sel_n is -1 (high impedance)
Rule 2: Reading
If *Data*(n) = 1 or 3 then
Strobe_n[n-d_R]=0, and
Write_Sel_n[n-d_R] =1
Rule 3: Writing
If *Data*[n] = 2 or 3 then
Strobe_n [n-d_W] = 0, and
Write_Sel_n [n-d_W] = 0
Where d_R and d_W are delay cycles of reading and writing respectively.

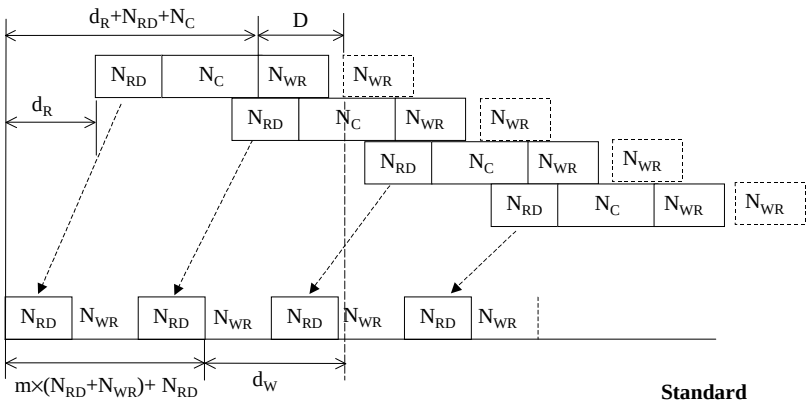
Memory Access Control Scheduling

For each index *n* of *data* do
Strobe_n[*n*]← 1 (default logic value, high)
Write_Sel_n[*n*] ← -1 (default logic value, high impedance)
For *n* ← first index of *data* to the last index of *data* do
If *data*[*n*] = 1 or 3 then //reading
If *Write_Sel_n*[*n*- d_R] = 0 then Return "**memory access conflict**";
Else *Strobe_n*[*n*- d_R] = 0; *Write_Sel_n*[*n*- d_R] = 1
If *data*[*n*] = 2 or 3 then //writing
If *Write_Sel_n*[*n*- d_W] = 1 then Return "**memory access conflict**";
Else *Strobe_n*[*n*- d_W] = 0; *Write_Sel_n*[*n*- d_W] = 0

Example:
d_R=2, d_W=0 and
data=(1, 2, 1, 2)

Index	-2	-1	0	1	2	3
Strobe_n	0	1	0	0	1	0
Write_En_n	0	-1	0	1	0	1
data			1	2	1	2

Loop Pipelining with Modulo Schedule



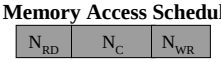
prologue number

$$m = \left\lceil \frac{N_C - d_W + d_R}{N_{RD} + N_{WR}} \right\rceil$$

The delay cycles of the write operation

$$D = m\times(N_{RD} + N_{WR}) - d_R - N_C + d_W$$

Standard Memory Access Schedule



$$II_{MIN} = N_R + N_W$$

Algorithm

Generate the memory access control schedule of pipelined loop

(1) $II=N_R+N_W$
(2) Compute the prologue number *m*.
(3) Compute the delay cycles of the write operations *D*.
(4) Shift all writing operations in *data* to the right by D, i.e.
    data = RSH($N_{RD}+N_C$,D,*data*)
(5) Shift and add schedules
    data_s = *data*;
    For i=1 to *m* do
        data = *data_s* + RSH(0, II, *data*);
(6) Perform Memory Access Control Scheduling for *data*
(7) Number of execution cycles of prologue: II×*m*.
    Number of execution cycles of steady state: II×($N_{ITER} - m$)
    Number of execution cycles of epilogue: II×*m*

Representation

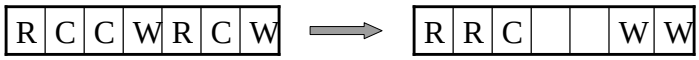
data=(6,3,2)
data=(1,1,1,1,1,1,0,0,0,2,2)
N_{RD}=6, N_C=3 and N_W=2.

Shift Operation

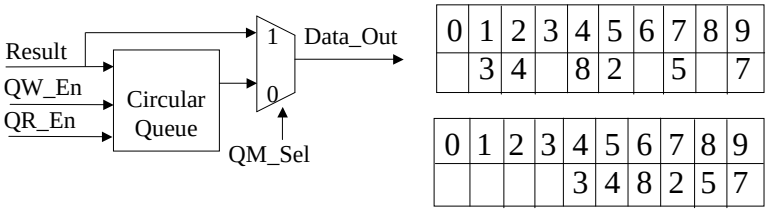
RSH(*index*, *distance*, *data*)

RSH(9,1,*data*) =
(1,1,1,1,1,1,0,0,0,0,2,2)

Standardizing Memory Access Schedule



• May need to add a Circular Queue and a Multiplexer



• What are the shortest length of the circular queue?

Algorithm

Compute *QW_En, QR_En, QM_Sel,* and the length of the circular queue

Two scans of the original schedule.

The first scan

```
N_k = 0; flag = 1; L_MIN = 0;
For n ← N-1 downto 0 do
   QW_En[n] = 0;
   QM_Sel[n] = 0;
   If flag = 1 then
      If data[n] = 2 or 3 then
         QM_Sel[n] = 1; N_k++;
      Else flag = 0;
   Else
      If data[n] = 2 or 3 then
         QW_En[n] = 1; L_MIN++;
```

L_{MIN}: # of data in the queue

N_k: # of data not in the queue

The second scan

```
num = L_MIN;
For n ← N-1 downto N-N_k do
   QR_En[n] = 0;
For n ← N-N_k-1 downto N-N_k-num do
   QR_En[n] = 1;
   If data[n] = 2 or 3 then
      L_MIN --;
For n ← N-N_k-num-1 downto 0 do
   QR_En[n] = 0;
```

L_{MIN}: The minimum length of queue

Conclusions

- The results can be extended to the case of multiple memory ports
- The technique simplifies the loop computation synthesis
- The technique enables to seek an “optimal” FPGA design for a nested loop computation.
- The algorithms enable to generate a memory controller for a pipelined loop computation automatically. It has been applied to an automated FPGA design tool at Wright State University