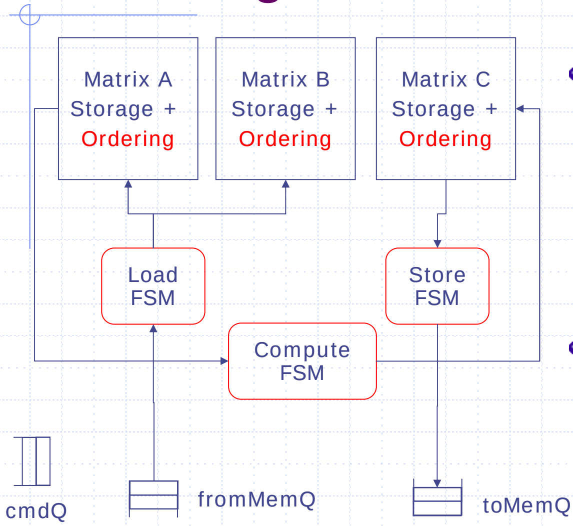
Matrix Multiply: Writing and Refining FSMs

Nirav Dave
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Revising the MAC Unit



- FSMs are tied to resources
  - No structural hazards
  - FSMs can run concurrently
  - addresses computed in the storage
- Data dependences?
  - Add logic to storage to keep reads and writes in the correct order

New Storage IFC

```
interface Storage#(type data)
 method Action startRead(RCmd c);
 method ActionValue#(Tuple2#(Bool,data))
                     read();
 method Action startWrite(WCmd C)
 method ActionValue#(Bool) write(data x);
endinterface
```

What access pattern to read?

Is this the last read?

What access pattern to write?

Is this the last write?

Starting Commands Revisited

```
rule startCommand(True);
 commandQ.deq();
 case commandQ.first()
    LoadA: loadFSM.start(); loadQ.enq(A); a.startWrite(Linear);
    LoadB: loadFSM.start(); loadQ.enq(B); b.startWrite(Linear);
    StoreC: storeFSM.start(); storeQ.enq(C);
                              c.startRead(Linear);
    MultAccum: a.startRead(MatrixIK);
               b.startRead(MatrixJK);
               c.startRead(MatrixIJ);
               c.startWrite(MatrixIJ);
               compQ.enq(MAC); multFSM.start();
    Mult:      a.startRead(MatrixIK);
               b.startRead(MatrixJK);
               c.startWrite(MatrixIJ);
               compQ.enq(MUL); multFSM.start();
 endcase
endrule
```

We've told the storage how what access pattern and whether which operations happen first (reads or writes)

Matrix{1}{2}: Count up {i,j,k} adding and use the address {1,2}

The Load FSM

```
Stmt load = seq
 doneLoad <= False;
 while (!doneLoad)
  action
   let loadM = (loadQ.first == A) ? a : b;
   let doneWrite <- loadM.write(fromMemQ.first());
   fromMemQ.deq();
   doneLoad <= doneWrite;
  endaction
 loadQ.deq();
endseq;
```

This is almost a single-action loop: Can we rewrite it as a rule?

Yes

Starting Commands Revisited

```
rule startCommand(True);
 commandQ.deq();
 case commandQ.first()
    LoadA: loadQ.enq(A); a.startWrite(Linear);
    LoadB: loadQ.enq(B); b.startWrite(Linear);
    StoreC: storeQ.enq(C); c.startRead(Linear);
    MultAccum: a.startRead(MatrixIK);
               b.startRead(MatrixJK);
               c.startRead(MatrixIJ);
               c.startWrite(MatrixIJ);
               compQ.enq(MAC);
    Mult:      a.startRead(MatrixIK);
               b.startRead(MatrixJK);
               c.startWrite(MatrixIJ);
               compQ.enq(MUL);
 endcase
endrule
```

We've told the storage how what access pattern and whether which operations happen first (reads or writes)

Matrix{1}{2}: Count up {i,j,k} adding and use the address {1,2}

enqs implicitly starts the "FSM"

The Load/Store "FSM"

```
rule doLoad(True);
 let loadM = (loadQ.first == A) ? a : b;
 let doneWrite <- loadM.write(fromMem.first());
 fromMem.deq();
 if (doneWrite) loadQ.deq();
endrule


rule doStore(True);
 let {.doneRead, .readVal} <- c.read();
 toMem.enq(readVal);
 if (doneRead) storeQ.deq();
endrule
```


The Compute "FSM"

```
rule compute(True);
 let {.doneA, av} <- a.read();
 let {.doneB, bv} <- b.read();
 let {.doneC, cv} <- (compQ.first()==MAC)
                   ? c.read()
                   : return(tuple2(doneA, 0));
 let finished <- c.write(cv + av*bv);
 if (finished) compQ.deq();
endrule
```

The rules are very simple. We've moved the complexity into the storage units.

Implementing a Storage Unit

```
interface Storage#(data)
 method Action startRead(RCmd c);
 method ActionValue#(Tuple2#(Bool,data))
               read();

 method Action startWrite(WCmd c);
 method ActionValue#(Bool) write(data x);
endinterface
```


Storage Internal State

```
Reg#(RCmd) rdPattern <- mkRegU;
Reg#(Bool) rdActive  <- mkReg(False);
Reg#(WCmd) wrPattern <- mkRegU;
Reg#(Bool) wrActive  <- mkReg(False);
Reg#(Bool) rdIsOlder <- mkReg(True);

Reg#(Bit#(three_n)) rdIJK <- mkReg(0);
match {.rdI,.rdJ,.rdK} = split3(rdIJK);

Reg#(Bit#(three_n)) wrIJK <- mkReg(0);
match {.wrI,.wrJ,.wrK} = split3(wrIJK);

RegFile#(Bit#(two_n), data) mem <- mkRegFileFull();
```

Starting reads and writes

```
method Action startRead(RCmd cmd) if (!rdActive)
  rdActive  <= True;
  rdPattern <= cmd;
  if (wrActive) rdIsOlder <= False;
endmethod

method Action startWrite(WCmd cmd) if (!wrActive)
  wrActive  <= True;
  wrPattern <= cmd;
  if (rdActive) rdIsOlder <= True;
endmethod
```

Issue: Compiler doesn't determine these can be done in parallel (though they are safe). Tweaking is required

Read

```
Bool canRead = <more later>

method Actionvalue#(Tuple#(Bool,data))
                     read() if(canRead);
  rdIJK <= rdIJK + 1;
  Bool done = (rdIJK == {maxN,maxN,maxN} ||
     (rdIJK == {  0,maxN,maxN} && rdPattern==Linear));
  let rdAddr = case(rdPattern)
              MatrixIK: return {rdI,rdK};
              MatrixJK: return {rdJ,rdK};
              default:  return {rdI,rdJ}; endcase;
  if (done) rdActive <= False;
  return tuple2(done, mem.sub(rdAddr));
endmethod
```

Write

```
Bool canWrite = <more later>

method Actionvalue#(Bool)
                     write(data val) if(canWrite);
  wrIJK <= wrIJK + 1;
  Bool done = (wrIJK == {maxN,maxN,maxN} ||
     (wrIJK == {  0,maxN,maxN} && wrPattern==Linear));
  let wrAddr = case(wrPattern)
                MatrixIK: return {wrI,wrK};
                MatrixJK: return {wrJ,wrK};
                default:  return {wrI,wrJ}; endcase;
  if (done) wrActive <= False;
  mem.upd(wrAddr, val);
  return done;
endmethod
```


CanRead/CanWrite

- We read in linear order (in the major order for the matrix)
  - Repeat reads
- Given the place in compute, we can determine:
  - The next address we need to access
  - The largest address we don't need to access again
- canRead = "only reading" or "read is older" or "write has finished writing the next address to be read
- Similar for canWrite

A bit complicated to get the logic correct, but the idea is simple

What happened to the FSMs?

- FSMs are good at:
  - Dealing with structural hazards
  - Representing interlocks between actions
- Out changes to the microarchitecture:
  - Removed structural hazards (reorganized FSMs)
  - Made interlocks explicit
- Removed the value of using the FSMs construct
  - rules become more natural
- Isolated the FSM tasks until nothing was left

Current Design

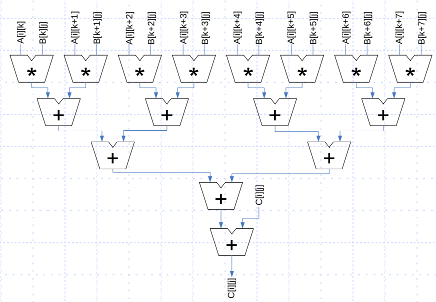
- Our design is pretty efficient
  - Starts compute as soon as possible
  - All the rules are local (circuit quality)
  - No extraenous state
- But we still only handle 1 multiply/cycle
  - Let's try to do multiple steps at once?

Acheiving HW Parallelism

- Obvious solution: Unroll the loop P times

```
for(int i = 0; i < K; i++)
 for(int j = 0; j < K; j++)
  for(int k = 0; k < K; k++)
     c[i][j] +=
        a[i][k] * b[k][j];
```

- Adder Tree of P
  - Crit. path grows logarithmically
  - Limited parallelism before we hurt cycle delay



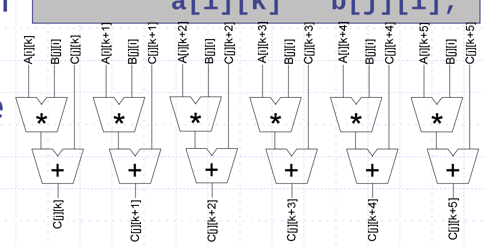
A Better Pattern

- Reorder loops
  - Increment **c[i][j]** multiple times

```
for(int i = 0; i < K; i++)
 for(int j = 0; j < K;
j++)
  for(int k = 0; k < K;
k++)
     c[j][k] +=
        a[i][k] * b[j][i];
```

- Shorter critical path

- Need to change the storage access patterns



Simple Optimizations:

- Matrix Storage A, B, and C only need a few of the existing access patterns.
  - Specialize the implementations for the necessary ones
- Multiply-Add in MAC Unit is the critical path
  - Pipeline the computation
- Replace RegFile with BRAM
  - single-cycle read delay
  - Better FPGA area properties

FSM Cycle-Level Performance

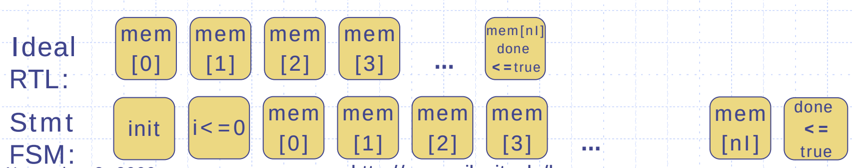
- Stmt FSMs are slower than ideal

Hand-tuned FSM

```
rule initialize (i < nI);
  mem.write (addr, f(i));
  addr <= addr + 1;
  i <= i + 1;
  if (i+1 == nI)
     done<=True;
endrule
```

Stmt FSM

```
Stmt s = seq
 for(i<=0; i<nI; i<=i+1)
  action
   mem.write(addr,f(i));
   addr <= addr + 1;
  endaction;
 done <= True;
endseq;
```



Performance Doubly Nested Loops?

- Lose 1 cycle per loop iteration
- Is this okay?
- How do we remove the dead cycle?

```
Stmt s = seq
 for(i<=0; i<nI; i<=i+1)
  for(j<=0; j<nJ; j<=j+1)
   action
    mem.write(addr,f(i));
    addr <= addr + 1;
  endaction;
 done <= True;
endseq;
```


Lab 5!

- Lab 4 – SMIPv2 Processor
  - Improved compute performance
    - pipelining
    - branch prediction
- In Lab 5 we will improve the memory system
  - only the instruction cache (no coherence issues)

Memory System

- Lab 4 ignored the cold misses
  - Ran statistics after warming the cache
- Your task: non-blocking caches
  - Start the lookup for the 2nd miss before you have received the data for the 1st

Important Concerns:

- Do your cache responses come back in-order?
  - What happens when you miss then hit?
- Do you always return the correct value?
- How fast are cache hits?
  - Is it as fast as in Lab 4?
- How do you deal with requests from mispredicted instructions?
  - Is it fast?
- How many requests can you issue concurrently?
  - Where do you need buffering?