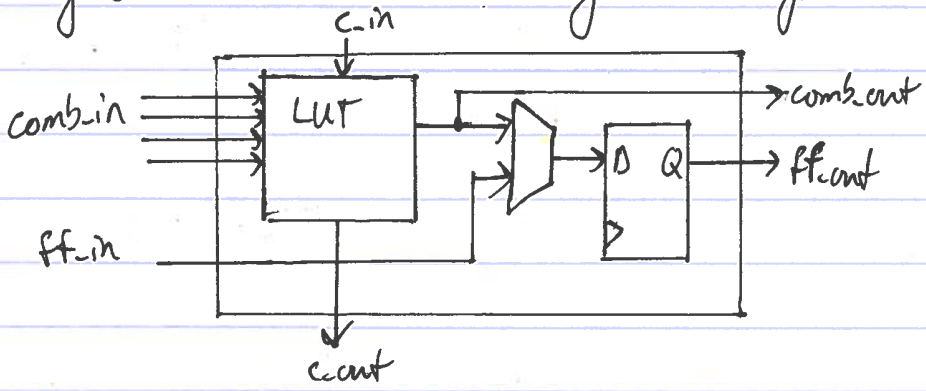
High-Level Synthesis

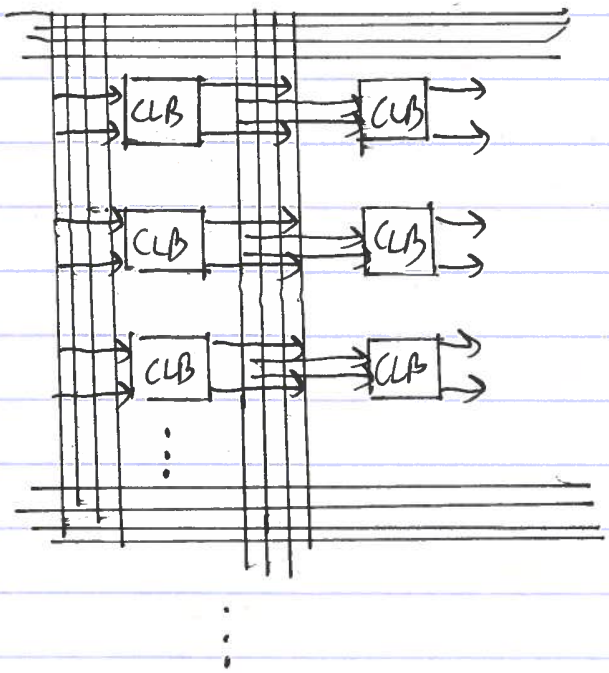
HLS converts C/C++/SystemC into RTL (register transfer level) designs
- it maps data structures, operations and communication onto hardware blocks (Verilog, VHDL)
- a standard synthesis tool then converts the RTL design to an FPGA bitstream or ASIC masks

FPGAs

- grid of CLBs (configurable logic blocks)



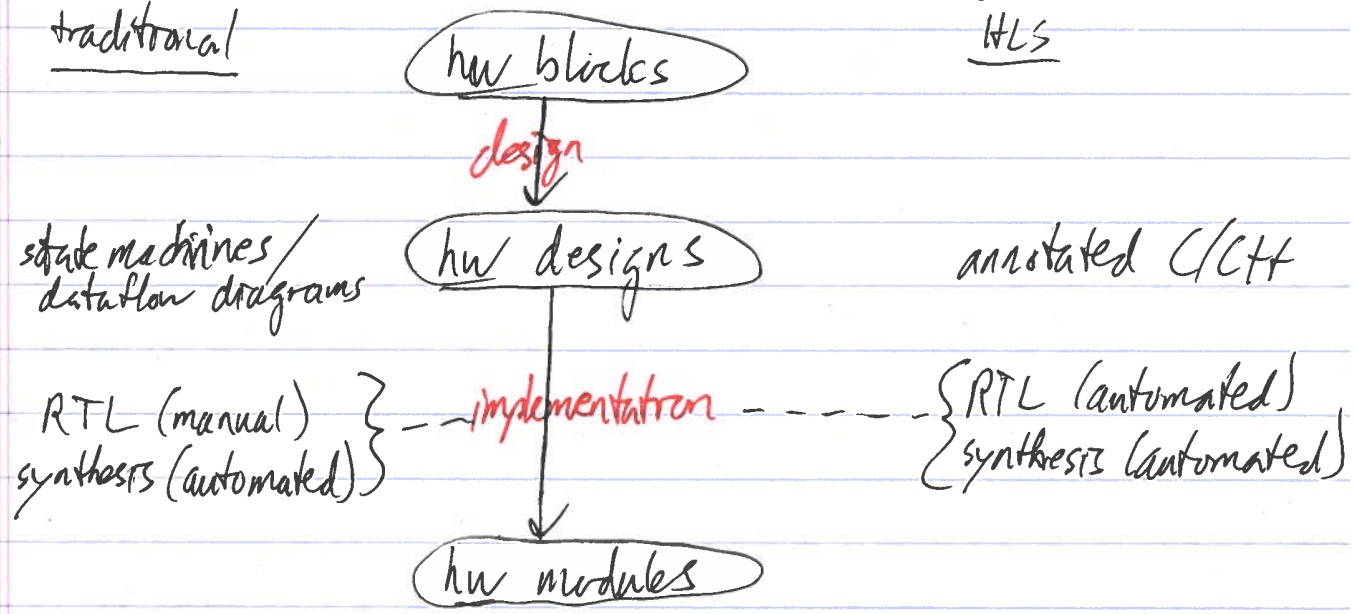
ff = flip-flop
lut = look-up table
- implements a boolean function



- each column contains
  - CLBs or
  - DSP blocks or
  - block RAM (BRAM) (dual-ported SRAM)

- FPGA may also contain:
  - memory interfaces (e.g. DDR)
  - I/O interfaces (e.g. PCIe)
  - processors

HLS can replace the traditional hw design flow



pros:
- C/C++ spec. more compact than the RTL
- quicker exploration of design alternatives
  e.g. pipelining - require a rewrite in RTL but only takes 1 annotation in HLS
- good at optimizing datapath
- generates interfaces automatically

cons:
- not so good at optimizing control flow (e.g. complex functions)
- limitations on C/C++ used
- need to understand the tool to generate an efficient design

- summary: increases productivity but is a hw designer's tool, not a sw designer's tool

HLS transforms a function into a hw IP block
- each nested function call generates a hw module (unless the function is inlined)
- multiple nested calls may result in multiple instances (parallelism)
- invoking the function is turned into triggering the hw block (known as a "transaction")

Restrictions on C/C++ for HLS
- no recursion - no call stack
  - instead, translate recursive algorithms into iterative algorithms
- avoid pointers; some simple pointer arithmetic can be handled though
  e.g. iterating over a static array is okay;
  manipulating dynamic linked lists and trees not okay

① Use constant array sizes
- the HLS tool needs to determine memory size
- if size must depend on input, then declare upper bound
  e.g. assert(size < CONST); (actually use compiler directives in Vitis HLS)

(2) Use constant loop bounds
- needed to determine latency
- if the # of iterations depends on input, ~~use~~ declare upper bound

```
assert(k < CONST);
for(i=0; i<k; i++) ...
```

- can handle simple cases such as

```
e.g for(i=0; i< CONST; i++)
        for(j=i; j< CONST; j++)
            foo(i,j);
```

iterations of foo() = $\sum_{j=1}^{CONST} j = \frac{(CONST)(CONST+1)}{2}$

(3) Avoid branches that can't be "flattened"
- branches: if or switch
- flattening: compute both/all paths and use multiplexer to choose the output
- difficult to flatten if alternate paths have different latencies

```
e.g for(i=0; i< CONST; i++)
        if(a[i] <0)
            a[i] *=2;
        else
            a[i] *= a[i];
```

same latency, therefore okay

e.g.
```
if(a<0)
    for(i=0; i< CONST1; i++)
        foo(i);
else
    for(i=0; i< CONST2; i++)
        oof(i);
```

- this block's latency is variable which makes the RTL hard to optimize
- can be use but may have poor performance

passed by value (input only)

pass by reference (input/output)

Interfacing

e.g. Type3 toplevel(Type1 in, Type2 &inout) {

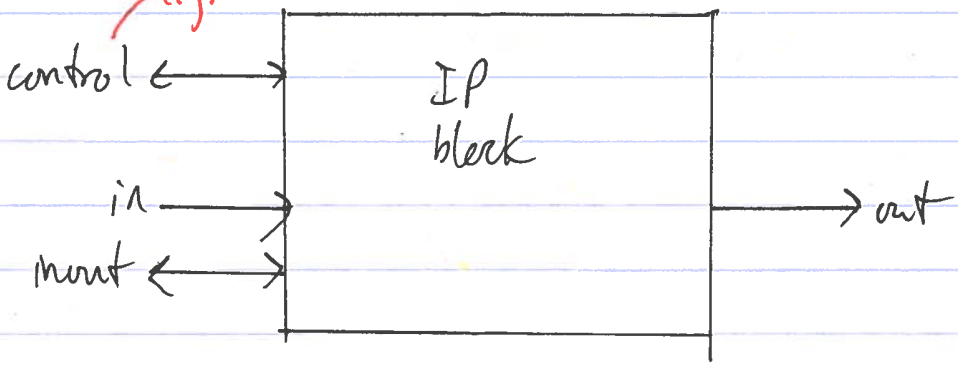
output → Type3 out;

...

return out;

}

e.g. start transaction, signal end



- control options:

tool generated

① implicit: transaction starts when inputs ready

② signalling: wires connected to other blocks

③ bus slave: block has memory-mapped control registers accessed by the processor (sw)

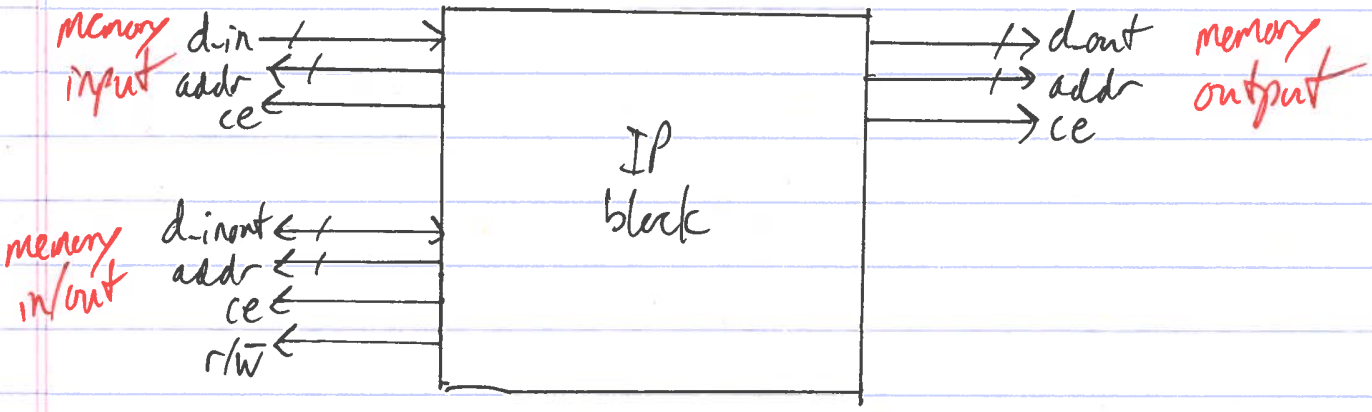
- end of transaction is communicated to sw by polling or interrupts

- input/output data (for the IP block)
  - interface type depends on the data

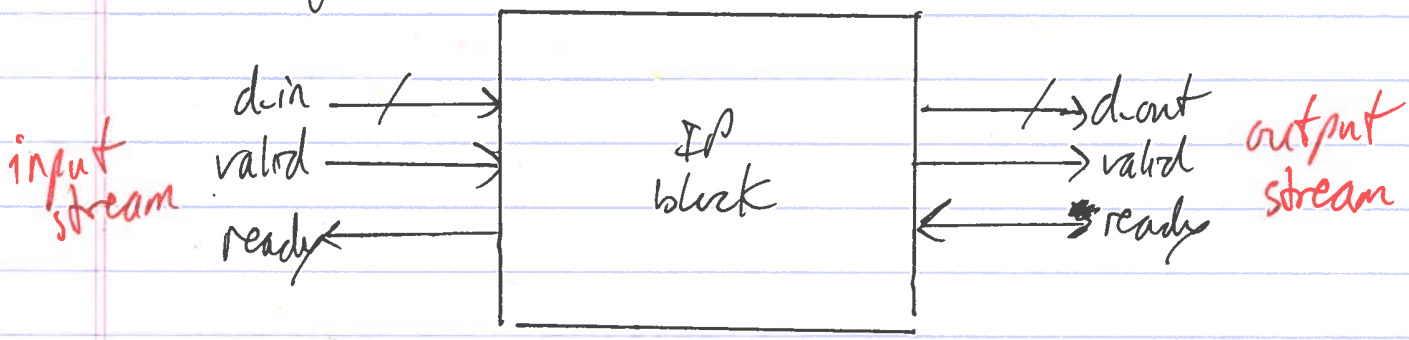
① fixed size (scalars, struct, class)
  a) data on wires from local memory (block RAM / BRAM)
  b) via memory-mapped data registers

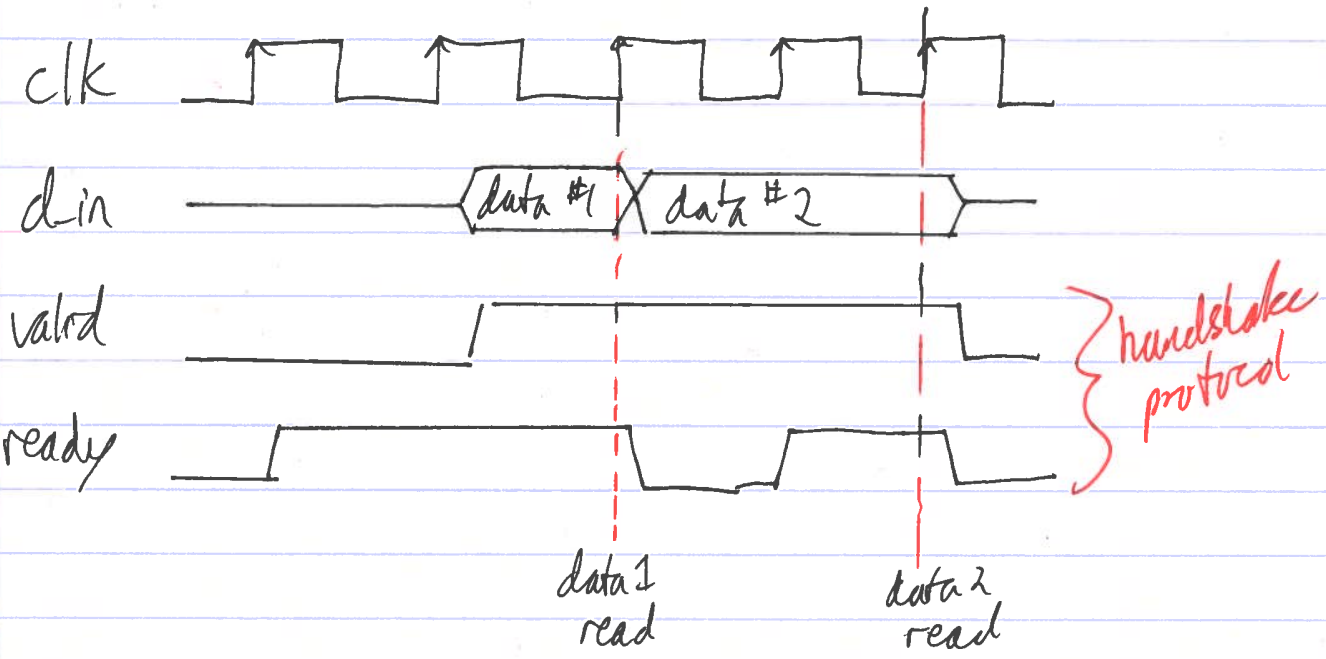
② arrays
  a) stored in local memory
    - global memory is off-chip (e.g. DDR / DRAM)
    - tool can generate load/store blocks to transfer between global memory and local memory
  b) bus master
    - IP block accesses global memory over the bus
    - global memory (DRAM) has high latency but good throughput for burst transfers
  c) streaming interface: global memory data is DMA'd to/from the block's streaming i/f
    - algorithm may need transforming to work on sequential data instead of random accesses or cache global data in local memory for random accesses

Memory Interface



Streaming Interface (e.g. AXI)





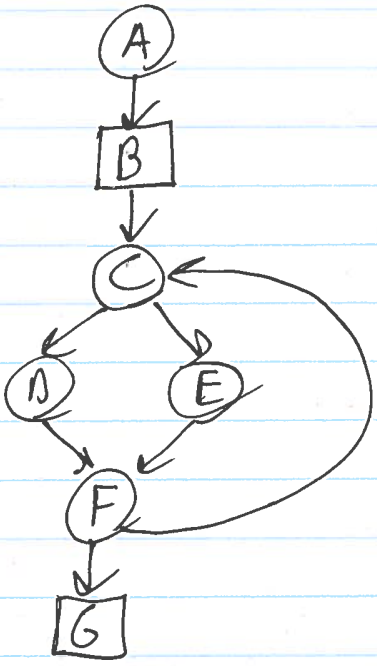
HLS continued

- hardware is inherently parallel
- types of parallelism (from the application)
  a) instruction level (datapath) — HLS ✓
  b) data level (loops) — HLS need guidance
  c) transaction level (data flow) — HLS ✓

Instruction Level Parallelism

① Control Flow Graph (CFG)
- nodes are basic blocks
- basic block:
  - has 1 entry point (top)
  - has 1 branch (bottom)

```
e.g.        a=0;                    A
            c=0;                    A
            x=rand();               B
loop:       if(x>0)                 C
              a+=x;                 D
            else
              a-=x;                 E
            c++;                    F
            if(c<CONST) goto loop;  F
            printf("%d", a);        G
```

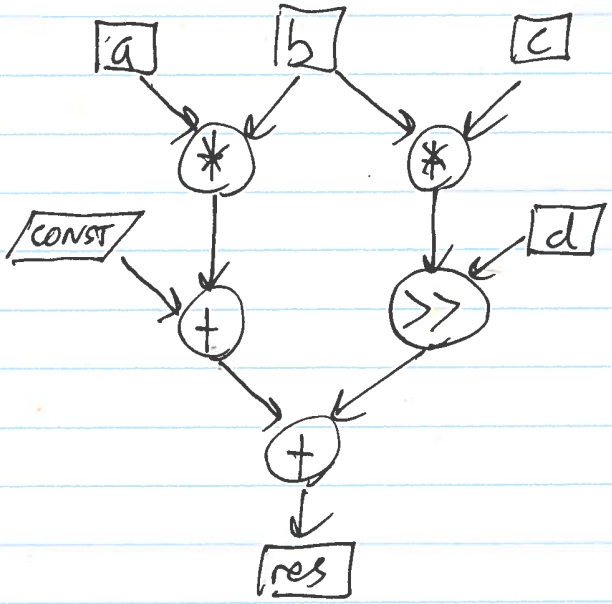


- function calls are treated as basic blocks

(2) Dataflow Graph (DFG)

- there is 1 DFG per basic block
- DFG represents data dependencies between operations

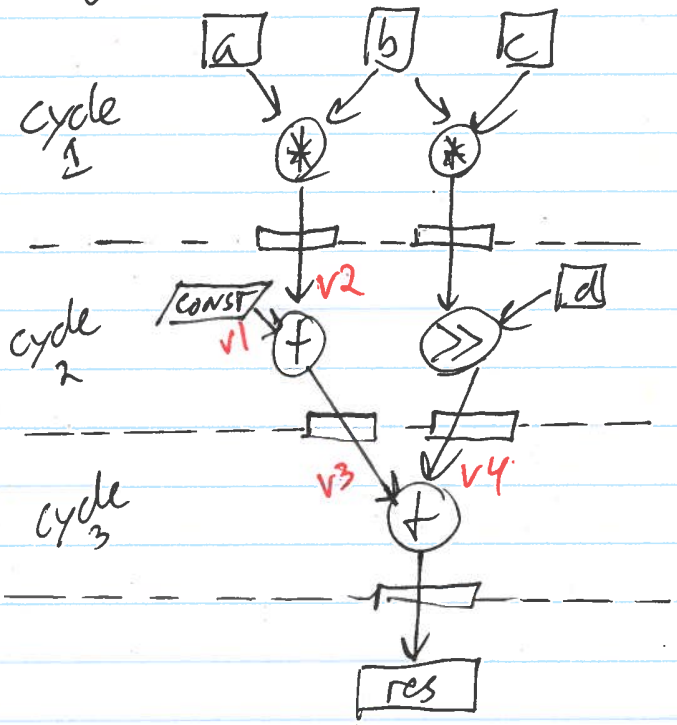
e.g.
```
int a, b, c, d;
const int CONST;
int tmp1 = a*b;
int tmp2 = b*c;
int res = tmp1 + CONST + (tmp2 >> d);
```



- registers are inserted in the DFG to shorten critical path length (create clocked stages)

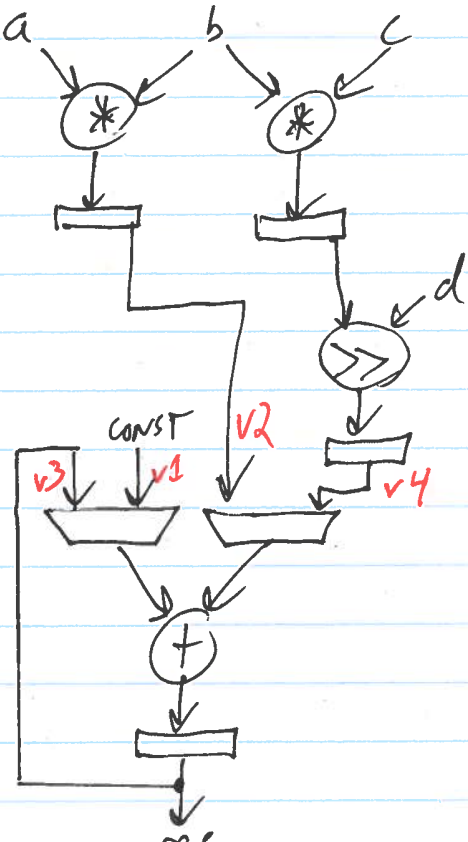
- datapath design steps:
  a) scheduling - assign operations to stages (clock cycles)
  b) mapping - assign operations to components

e.g. assume that *, +, >> all take 1 cycle



scheduled DFG

map to
2 multipliers
1 adder
1 shifter



hw implementation

- the multiplexers are ~~less~~ cost less area than a second adder
- they add a little latency but are probably not on the critical path (multiplier is likely slower than a multiplexer + adder)

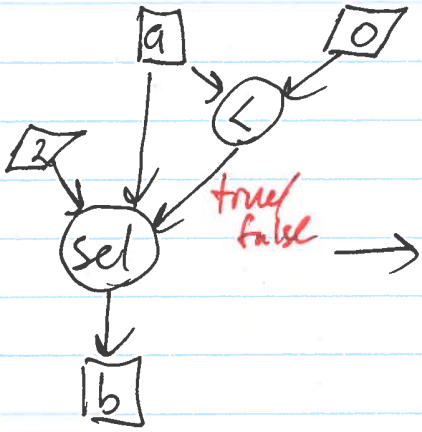
- to save hw area we could reuse 1 multiplier with a longer schedule (4 cycles)

Branch Flattening

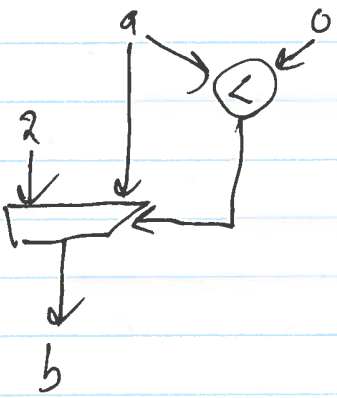
e.g.

BB1 { ⋮
if(a < 0)
BB2 b = 2;
else
BB3 b = a;
BB4 { ⋮

convert to selection node



flattened DFG



new implementation

- can't optimize across basic block boundaries

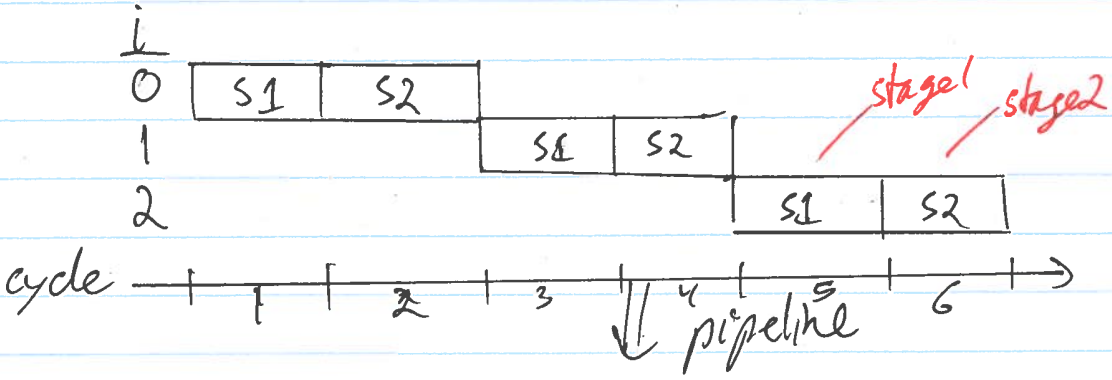
- branch flattening turns control flow into dataflow
- as a result the code above can be optimized as 1 basic block

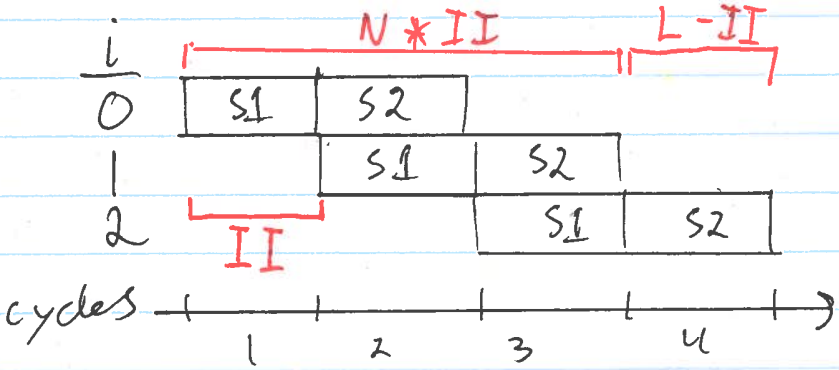
- try to structure branches so that they calculate values to be assigned to a variable - allows flattening which is important for achieving good performance

Data Level Parallelism

① Loop Pipeling

e.g. for(i=0; i<3; i++) N=3
BB; ← basic block of length L = 2 cycles





II (initialization interval) = 1

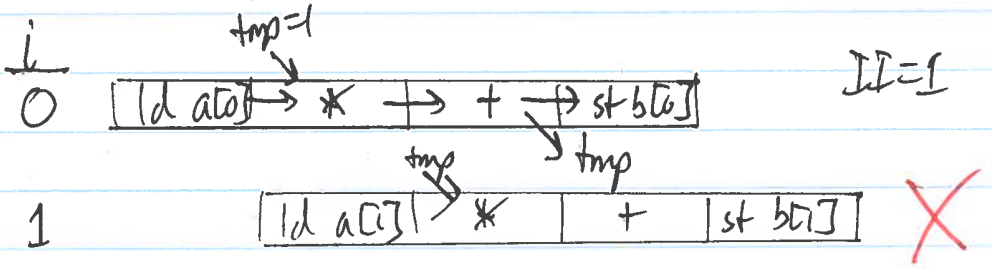
latency = N*II + L-II

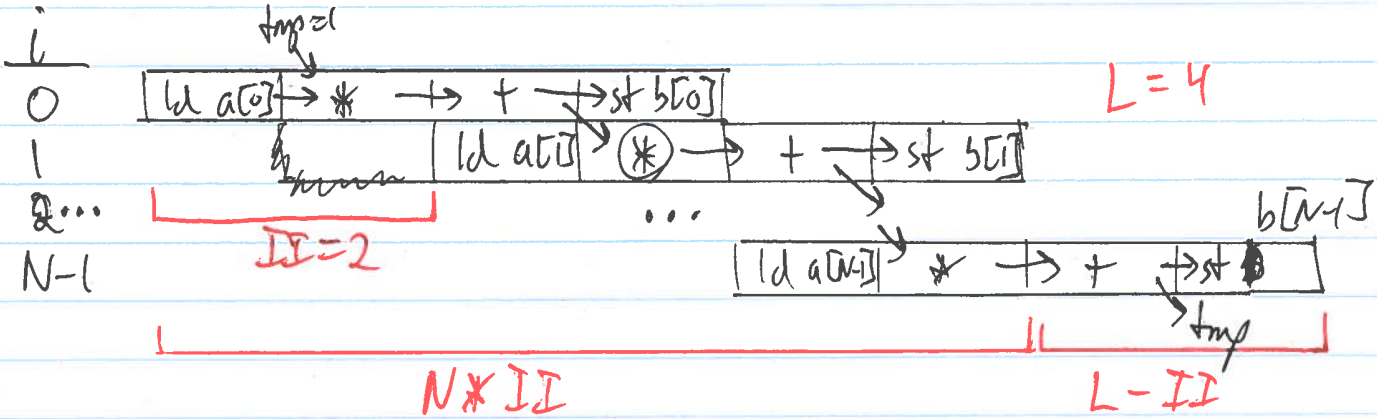
- pipelining requires more hardware (for II=1, 1 functional unit per operation)
- tool needs to be instructed to pipeline (not default behaviour)

- loop-carried dependencies (current iteration depends on previous iterations) increases the initialization interval (II)

e.g.
```
int a[N];
...
int tmp=1, b[N];
for(i=0; i<N; i++) {
    tmp = (tmp * a[i]) + CONST;
    b[i] = tmp;
}
```

- assumptions:
  - a[] & b[] are assigned to memory
  - tmp is assigned to a reg.
  - ld, st, *, + are all 1 cycle





latency = N*II + L-II

- flattening nested loops

e.g.
```
for(j=0; j<J; j++)
    for(k=0; k<K; k++)
        BB;          //length = L
```

- inner loop latency = ~~J*II + L-II~~ K*II + L-II
- overall latency = ~~K*(J*II + L-II)~~ J*(K*II + L-II)

- flattened loops:

```
for(jk = 0; jk < J*K; jk++) {
    calc j,k;   e.g. k = jk % K;  j = jk / K;
    BB;
}
```

- the ~~jk~~ j,k of the next iteration can be calculated in parallel with BB, so $\Delta L = 0$
- increases area cost

J× smaller

loop latency = J*K*II + L-II

- HLS compiler flattens nested loops automatically when pipelining

- if loops are not "perfectly nested", then it must be transformed before flattening

e.g.
```
for(j=0; j<J; j++) {
    BB1;
    for(k=0; k<K; k++)
        BB2;
}
```
can't flatten →

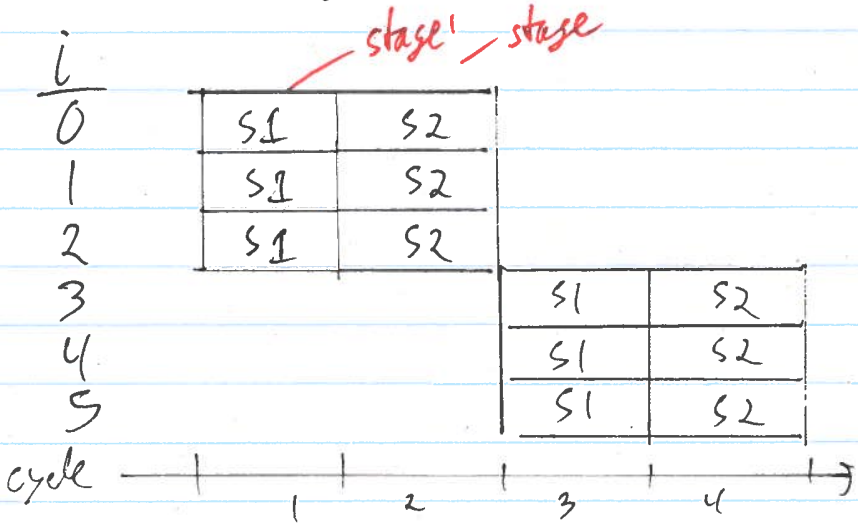
⇒

```
for(j=0; j<J; j++)
    for(k=0; k<K; k++){
        if(k==0) BB1;
        BB2;
}
```

- If BB1 and BB2 are independent (and $L_{BB1} \leq L_{BB2}$) then BB1 and BB2 can be executed in parallel
  - increases hw cost
- HLS compiler tries to do this too

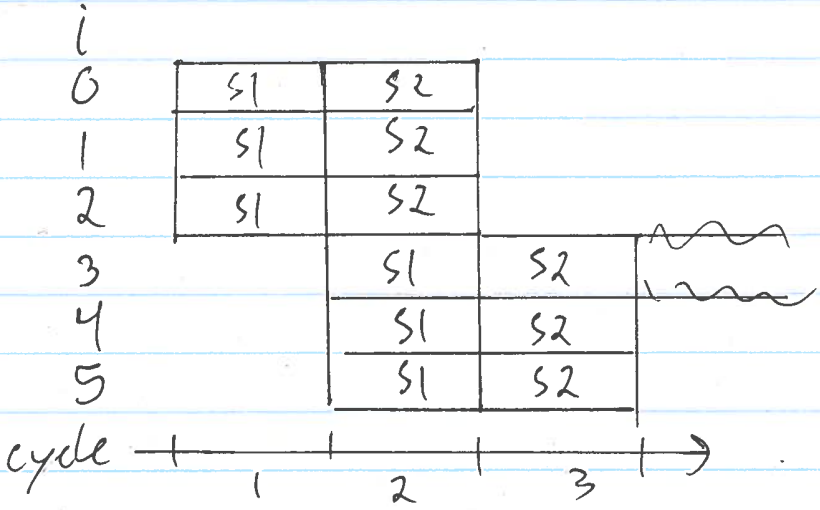
② Loop Unrolling

e.g. for(i=0; i<N; i++)
  BB; // L=2, unroll factor M=3
(assuming no loop-carried dependences)



latency = ⌈N/M⌉ * L
- increases area cost M times

Loop Unrolling and Pipelining

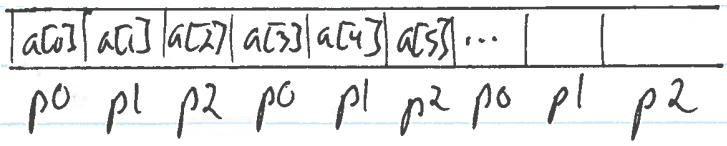


latency = ⌈N/M⌉ * II + L - II

- unrolling requires more interface/memory bandwidth

Partitioning

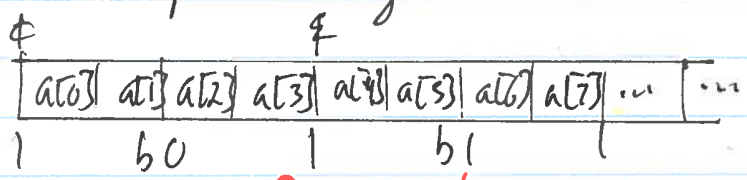
- partition an array into M sub-arrays
- each sub-array is in a different memory/interface
- cyclic partitioning: M=3



p0 = partition 0

- use if original accesses are sequential

- block partitioning: M=3

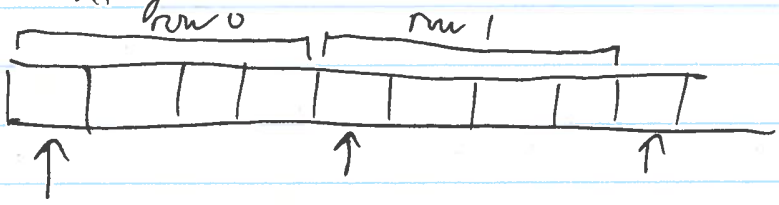


b0 = block 0

- use if original accesses are striding

- either way creates M memories/interfaces x sizeof(a[i])

striding



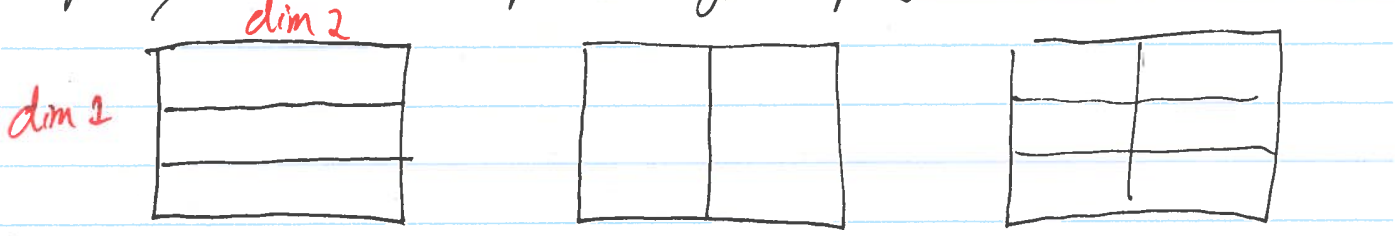

```
array[3][4];
for(i=0; i<3; i++)
  a[i][0] = x;
```

Reshaping

- creates one memory/interface x sizeof(a[i]) * M
- limited by ~~thing~~ BRAM width or max interface width

Matrices

- specify dimension for partitioning/reshaping



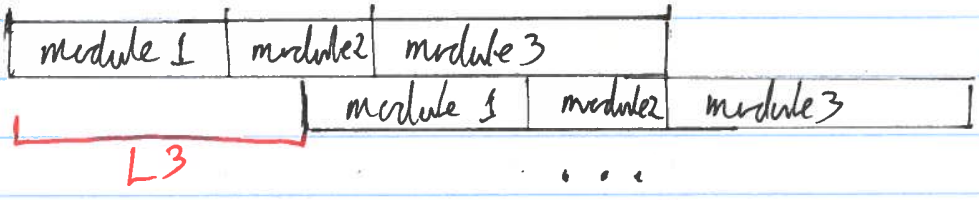
dim1: M=3 dim2: M=2 dim1: M=3, dim2: M=2

Transaction Level Parallelism (data flow parallelism)

e.g. void toplevel(...) {
    module1; //L1
    module2; //L2
    module3; //L3
}

— function call or loop

- pipelined transactions



- toplevel latency = L1 + L2 + L3
- interval = max(L1, L2, L3)
- doesn't improve the toplevel latency but does improve toplevel throughput

e.g. 24 fps

Matrix Multiplication Example

$R = A \times B$

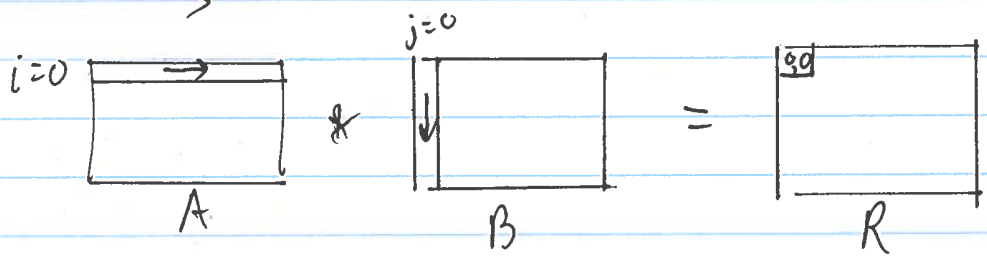
$I \times J$ — 16-bit signed ints

$I \times K$, $K \times J$ — 8-bit signed ints

- $I = J = K = 64$
- use streaming interfaces for each matrix
- target: Xilinx UltraScale+

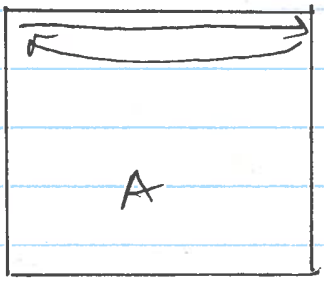
```
typedef int8_t a_t;
typedef int16_t r_t;

void multiply(a_t a[I][K], a_t b[K][J],
              r_t r[I][J]) {
Row:    for (int i=0; i<I; i++)
Col:        for(int j=0; j<J; j++) {
              r[i][j] = 0;
Product:      for(int k=0; k<K; k++)
                r[i][j] += a[i][k] * b[k][j];
            }
```



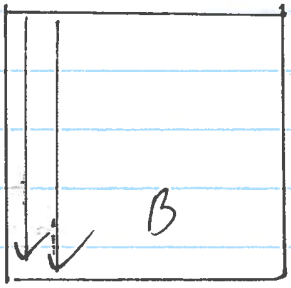
- note that matrices are stored in row-major order in C/C++
- this code can't be synthesized: streaming interface requires sequential accesses

- access patterns



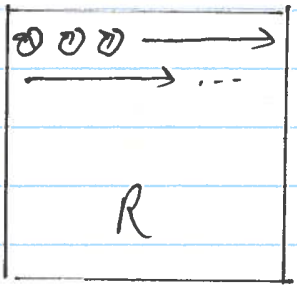
- repeat each row J times

input



- repeat whole matrix I times

input



- repeats each element K times

input/output

- first solution - make local copies of A, B, R (in BRAM)

Learn > Lecture > Handouts > matrix Handout

AXI4-streaming

Solution 1: local copies of each matrix

streaming interfaces

```
void multiply(a_b_t a[I][K], a_b_t b[J][K], r_t r[I][J]) {
#pragma HLS INTERFACE axis register both port=r
#pragma HLS INTERFACE axis register both port=b
#pragma HLS INTERFACE axis register both port=a

    a_b_t atmp[I][K], btmp [K][J];
    r_t rtmp[I][J];

Row_a_copy: for(int i=0; i<I; i++)
    Col_a_copy: for(int k=0; k<K; k++)
                    atmp[i][k] = a[i][k];

Row_b_copy: for(int k=0; k<K; k++)
    Col_b_copy: for(int j=0; j<J; j++)
                    btmp[k][j] = b[k][j];

Row: for(int i=0; i<I; i++)
    Col: for(int j=0; j<J; j++) {
            rtmp[i][j] = 0;
            Product: for(int k=0; k<K; k++)
                        rtmp[i][j] += atmp[i][k] * btmp[k][j];
        }

Row_res_copy: for(int i=0; i<I; i++)
    Col_res_copy: for(int j=0; j<J; j++)
                    r[i][j] = rtmp[i][j];

}
```

local copies

copy a

copy b

Synthesizable

output r

multiply:

Latency		Interval		
min	max	min	max	Type
553476	553476	553476	553476	none

sum

latency = trip count × iteration latency

loops:

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_a_copy	4224	4224	66	-	-	64	no
+ Col_a_copy	64	64	1	-	-	64	no
- Row_b_copy	4224	4224	66	-	-	64	no
+ Col_b_copy	64	64	1	-	-	64	no
- Row	532608	532608	8322	-	-	64	no
+ Col	8320	8320	130	-	-	64	no
++ Product	128	128	2	-	-	64	no
- Row_res_copy	12416	12416	194	-	-	64	no
+ Col_res_copy	192	192	3	-	-	64	no

Solution 2: pipeline all innermost loops

```
void multiply(a_b_t a[I][K], a_b_t b[J][K], r_t r[I][J]) {
#pragma HLS INTERFACE axis register both port=r
#pragma HLS INTERFACE axis register both port=b
#pragma HLS INTERFACE axis register both port=a

    a_b_t atmp[I][K], btmp [K][J];
    r_t rtmp[I][J];

Row_a_copy: for(int i=0; i<I; i++)
    Col_a_copy: for(int k=0; k<K; k++)
                    #pragma HLS PIPELINE
                    atmp[i][k] = a[i][k];

Row_b_copy: for(int k=0; k<K; k++)
    Col_b_copy: for(int j=0; j<J; j++)
                    #pragma HLS PIPELINE
                    btmp[k][j] = b[k][j];

Row: for(int i=0; i<I; i++)
    Col: for(int j=0; j<J; j++) {
            rtmp[i][j] = 0;
            Product: for(int k=0; k<K; k++)
                        #pragma HLS PIPELINE
                        rtmp[i][j] += atmp[i][k] * btmp[k][j];
        }

Row_res_copy: for(int i=0; i<I; i++)
    Col_res_copy: for(int j=0; j<J; j++)
                    #pragma HLS PIPELINE
                    r[i][j] = rtmp[i][j];

}
```

multiply:

Latency		Interval		
min	max	min	max	Type
548872	548872	548872	548872	none

loops:

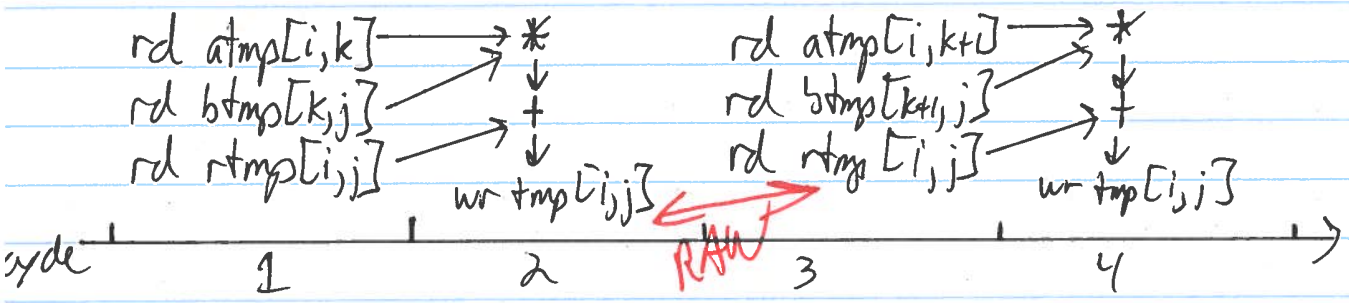
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_a_copy_Col_a_copy	4096	4096	1	1	1	4096	yes
- Row_b_copy_Col_b_copy	4096	4096	1	1	1	4096	yes
- Row_Col	536576	536576	131	-	-	4096	no
+ Product	128	128	2	2	1	64	yes
- Row_res_copy_Col_res_copy	4097	4097	3	1	1	4096	yes

- loops automatically flattened (saves 2 cycles)
- can't pipeline Product due to dependency

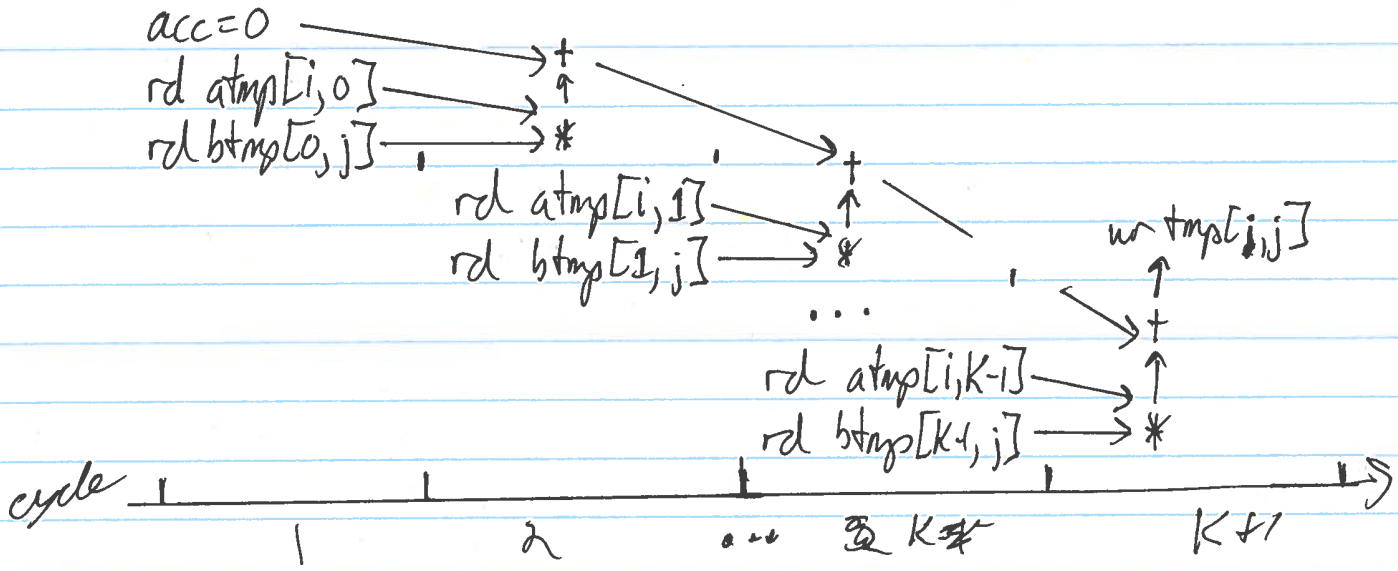
L II N

latency = N x II + L - II
= 4096 * 1 + (3-1) = 4098

Solution 2: can't unroll Product loop because of a RAW (read after write) dependency on rtmp[i][j]



Solution 3: use a register to accumulate value for r[i][j]



Solution 3: temporary register

```
Row: for(int i=0; i<I; i++)
    Col: for(int j=0; j<J; j++) {
            r_t acc = 0;
            Product: for(int k=0; k<K; k++)
                        #pragma HLS PIPELINE
                        acc += atmp[i][k] * btmp[k][j];
            rtmp[i][j] = acc;
        }
```

multiply:

Latency		Interval		
min	max	min	max	Type
274441	274441	274441	274441	none

loops:

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_a_copy_Col_a_copy	4096	4096	1	1	1	4096	yes
- Row_b_copy_Col_b_copy	4096	4096	1	1	1	4096	yes
- Row_Col_Product	262144	262144	2	1	1	262144	yes
- Row_res_copy_Col_res_copy	4097	4097	3	1	1	4096	yes

64×64×64 = 262144
I J K

Solution 4: unroll the Product loop

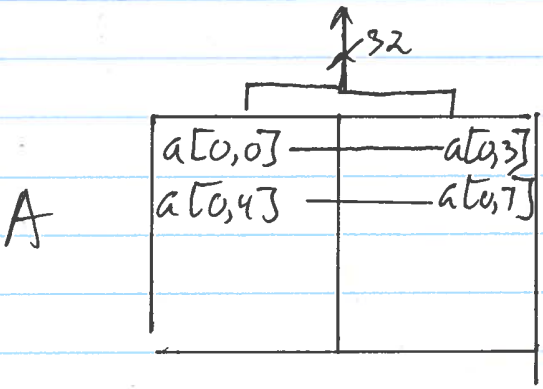
- each iteration of Product requires 1 read of A* and 1 read of B** (*atmp) (**btmp)
- unrolling will require more parallel reads of A and B and hence will require partitioning or reshaping
- our objective: determine the maximum unroll factor such that BRAM usage doesn't increase over Solution 3

- BRAMs: 18 kbits, 1-to-18 bit width, dual-ported (we could unroll with M=2 without partitioning or reshaping due to the dual-ported memory)

- existing BRAM usage:
  - size of A or B = 64 × 64 × 8 bits = 32 kbits
    ⌈32 kbit / 18 kbit⌉ = 2
  - size of R = 64 × 64 × 16 bits = 64 kbits
    ⌈64 kbit / 18 kbit⌉ = 4
  - Solution 3 uses 2 BRAMs of width 8 for each of A and B and 4 BRAMs of width 16 for R

access size

- try reshaping with factor of 4
  - width of A or B = 8 bits × 4 = 32 bits
    ⌈32 bit / 18 bit⌉ = 2 BRAM (to read 32 bits per cycle)



- no extra BRAMs required

- ~~width of R = 16 bits × 4 =~~ - no need to reshape R

- try reshaping with factor of 8
  - the number of BRAMs for A and B must double to allow 64 bit reads (per cycle)

- try ~~reshaping~~ partitioning with factor of 4
  - this requires 1 memory (~~BB~~ BRAM) per partition, so 4 BRAMs for each of A and B

- therefore, use reshape factor of 4
  - this allows 8 reads of A and of B per cycle because they are dual-ported

Solution 4: unroll M=8

```
void multiply(a_b_t a[I][K], a_b_t b[J][K], r_t r[I][J]) {
    #pragma HLS INTERFACE axis register both port=r
    #pragma HLS INTERFACE axis register both port=b
    #pragma HLS INTERFACE axis register both port=a

    a_b_t atmp[I][K], btmp [K][J];
    #pragma HLS ARRAY_RESHAPE variable=atmp cyclic factor=4 dim=2
    #pragma HLS ARRAY_RESHAPE variable=btmp cyclic factor=4 dim=1
    r_t rtmp[I][J];

    // copy A, copy B

Row: for(int i=0; i<I; i++)
    Col: for(int j=0; j<J; j++) {
        r_t acc = 0;
        Product: for(int k=0; k<K; k++)
            #pragma HLS PIPELINE
            #pragma HLS UNROLL factor=8
            acc += atmp[i][k] * btmp[k][j];
        rtmp[i][j] = acc;
    }

    // copy R

}
```

reshape A & B

sequential accesses

multiply:

Latency		Interval		
min	max	min	max	Type
57352	57352	57352	57352	none

loops:

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_a_copy_Col_a_copy	4096	4096	4	4	1	1024	yes
- Row_b_copy_Col_b_copy	4096	4096	1	1	1	4096	yes
- Row_Col	45056	45056	11	-	-	4096	no
+ Product	8	8	2	1	1	8	yes
- Row_res_copy_Col_res_copy	4097	4097	3	1	1	4096	yes

+3

kc 64/8 = 8

I J
64 x 64 = 4096

Row_Col latency = 4096 trips x 11 cycle/trip = 45056

(ideally it would be 4096 x 8 = 32768)

unrolling makes it hard to flatten

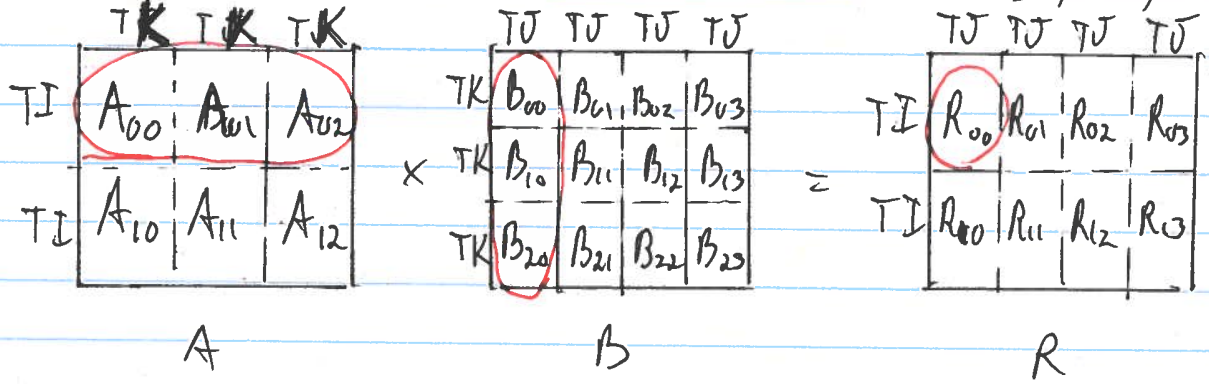
Solution 5

- further unrolling (eg. M=16) would require more parallel reads of A and B, leading to extra BRAM usage
- instead we can modify the algorithm to reduce repeated reads
- A and B are each 64×64 = 4k elements
  - each element is read 64 times: 256k reads for A and for B
- we could completely unroll the Row loop body
  - would result in reading 1 row of A at a time
  - would require 64×8 bits = 512 FF
  - but all of B still gets read every iteration
- we could unroll Row_Col_Prod
  - use a lot of functional units
  - would require 64×64×8 = 32k FF for each of A and B

Tiling

- divide the matrices into tiles with dimensions TI, TK, TJ

e.g.



- multiply by tiles row

e.g. $R_{00} = \underline{A_{00} \times B_{00}} + \underline{A_{01} \times B_{10}} + \underline{A_{02} \times B_{20}}$

each of these tile multiplies is a matrix multiplication which will be completely unroll

Solution 5: Tiling

TI = TK = TJ = 8

```
void multiply(a_b_t a[I][K], a_b_t b[J][K], r_t r[I][J]) {
    #pragma HLS INTERFACE axis register both port=r
    #pragma HLS INTERFACE axis register both port=b
    #pragma HLS INTERFACE axis register both port=a

    a_b_t atmp[I][K], btmp [K][J];
    #pragma HLS ARRAY_RESHAPE variable=atmp cyclic factor=4 dim=2
    #pragma HLS ARRAY_RESHAPE variable=btmp cyclic factor=4 dim=2
    r_t rtmp[I][J];
    #pragma HLS ARRAY_RESHAPE variable=rtmp cyclic factor=4 dim=2

    // copy A, copy B

TileRow: for(int ii=0; ii<I/TI; ii++)
    TileCol: for(int jj=0; jj<J/TJ; jj++)
        TileProduct: for(int kk=0; kk<K/TK; kk++)
            #pragma HLS PIPELINE
            Row: for(int i=0; i<TI; i++)
                Col: for(int j=0; j<TJ; j++) {
                    r_t acc = (kk == 0) ? 0 : rtmp[ii*TI + i][jj*TJ + j];
                    Product: for(int k=0; k<TK; k++)
                        acc += atmp[ii*TI + i][kk*TK + k] *
                               btmp[kk*TK + k][jj*TJ + j];
                    rtmp[ii*TI + i][jj*TJ + j] = acc;
                }

    // copy R

}
```

tile row index

tile col index

row index within a tile

completely unrolls inner loops

- reshape with factor 4
  ⇒ read 8 elements per cycle
- read 64 elements from A & B: 8 cycles
- tmp: (64 reads + 64 writes)/8 = 16 cycles

multiply:

Latency		Interval		
min	max	min	max	Type
20489	20489	20489	20489	none

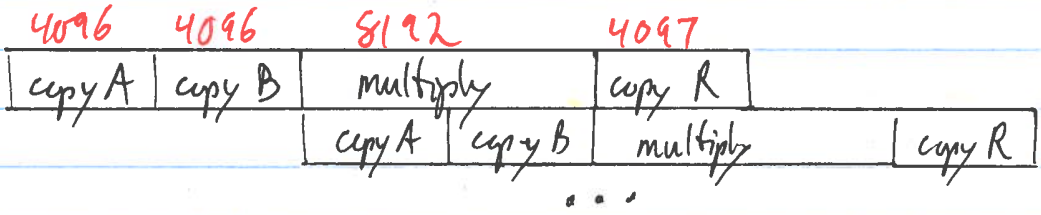
loops:

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_a_copy_Col_a_copy	4096	4096	4	4	1	1024	yes
- Row_b_copy_Col_b_copy	4096	4096	4	4	1	1024	yes
- TileRow_TileCol_TileProduct	8192	8192	16	16	1	512	yes
- Row_res_copy_Col_res_copy	4097	4097	6	4	1	1024	yes

8 × 8 × 8 (tiles)

tiling analysis
- # tiles per dimension = I/TI, J/TJ, K/TK
- before tiling (solution 4) # reads of A = I × J × K
- after tiling & unrolling, # reads of A
  = (I/TI × J/TJ × K/TK) × (TI × TK)
  # of tiles read — elements per tile
  = I × J/TJ × K
  - reduced reads of A by factor TJ
- analysis is similar for B

Solution 6: dataflow parallelism (transaction level)
— an invocation of the top-level module



- see Solution 6 handout

- since the A and B dimensions are the same, we could fuse copy A and copy B

e.g.
```
for(r=0; r<64; r++)
    for(c=0; c<64; c++) {
        #pragma HLS PIPELINE
        atmp[r][c] = a[r][c];
        btmp[r][c] = b[r][c];
```

- would reduce transaction latency (by 4096) but not the initiation interval (= 8192)

Solution 6: Dataflow

```
void multiply(a_b_t a[I][K], a_b_t b[J][K], r_t r[I][J]) {
    #pragma HLS INTERFACE axis register both port=r
    #pragma HLS INTERFACE axis register both port=b
    #pragma HLS INTERFACE axis register both port=a

    #pragma HLS DATAFLOW
    ...tiling solution...
}
```

multiply:
interval = 8195 = 8192 (multiply block) + 3 cycles

Summary

each DSP can do 2 multiplies

Solution	Interval	Clock Period ns	BRAM	DSP	FF	LUT
1	553,476	3.770	8	1	264	695
2	548,872	5.007	8	1	274	918
3	274,441	5.007	8	1	285	1052
4	57,352	6.466	8	4	332	1581
5	20,489	7.733	8	320	7,955	13,579
6	8,195	8.586	8	320	8,751	13,672

extra FFs to hold stage outputs

- results are estimates – not synthesized
- unrolling M=8 performs 8 multiplications in parallel
- tiling performs 8x8x8=512 multiplications in parallel