Chapter 4
Software Optimisation

Part 1: Optimisation Methods.

Part 2: Software Pipelining.

Chapter 4

Software Optimisation

Part 2 - Software Pipelining

Objectives

- Why using Software Pipelining, SP?
- Understand software pipelining concepts.
- Use software pipelining procedure.
- Code the word-wide software pipelined dot-product routine.
- Determine if your pipelined code is more efficient with or without prolog and epilog.

Why using Software Pipelining, SP?

- SP creates highly optimized loop-code by:
  - Putting several instructions in parallel.
  - Filling delay slots with useful code.
  - Maximizes functional units.
- SP is implemented by simply using the tools:
  - Compiler options -o2 or -o3.
  - Assembly Optimizer if .sa file.

Software Pipeline concept

To explain the concept of software pipelining, we will assume that all instructions execute in on cycle.

```
      LDH
||    LDH
      MPY
      ADD
```

How many cycles would it take to perform this loop 5 times?

(Disregard delay-slots).

______________ cycles

Software Pipeline Example

```
     LDH
||   LDH
     MPY
     ADD
```

How many cycles would it take to perform this loop 5 times?

(Disregard delay-slots).

5 x 3 = 15
__________________ cycles

Let's examine hardware (functional units) usage ...

Non-Pipelined Code

Cycle	.D1	.D2	.M1	.M2	.L1	.L2	.S1	.S2
1	ldh	ldh						
2			mpy					
3					add			
4	ldh	ldh						
5			mpy					
6					add			
7	ldh	ldh						
8			mpy					
9					add			

Pipelining Code

Cycle								
1	ldh	ldh	.M1	.M2	.L1	.L2	.S1	.S2
2	ldh	ldh	mpy					
3	ldh	ldh	mpy		add			
4	ldh	ldh	mpy		add			
5	ldh	ldh	mpy		add			
6			mpy		add			
7					add			

Pipelining these instructions took 1/2 the cycles!

Pipelining Code

Cycle								
1	ldh	ldh	.M1	.M2	.L1	.L2	.S1	.S2
2	ldh	ldh	mpy					
3	ldh	ldh	mpy		add			
4	ldh	ldh	mpy		add			
5	ldh	ldh	mpy		add			
6			mpy		add			
7					add			

Pipelining these instructions takes only 7 cycles!

Pipelining Code

Prolog

Staging for loop.

Loop Kernel

Single-cycle “loop” iterated three times.

Epilog

Completing final operations.

1	ldh	ldh	.M1	.L1
2	ldh	ldh	mpy	
3	ldh	ldh	mpy	add
4	ldh	ldh	mpy	add
5	ldh	ldh	mpy	add
6			mpy	add
7				add

Pipelined Code

```
prolog:         LDH    ; load 1
           ||   LDH

                MPY    ; mpy  1
           ||   LDH    ; load 2
           ||   LDH
loop:           ADD    ; add  1
           ||   MPY    ; mpy  2
           ||   LDH    ; load 3
           ||   LDH

                ADD    ; add  2
           ||   MPY    ; mpy  3
           ||   LDH    ; load 4
           ||   LDH
                .
                .
```

Software Pipelining Procedure

1. Write algorithm in C code & verify.
2. Write ‘C6x Linear Assembly code.
3. Create dependency graph.
4. Allocate registers.
5. Create scheduling table.
6. Translate scheduling table to ‘C6x code.

Software Pipelining Example (Step 1)

```
short DotP(short *m, short *n, short count)
{ int i;
  short product;
  short sum = 0;
  for (i=0; i < count; i++)
  {
    product = m[i] * n[i];
    sum += product;
  }
  return(sum);
}
```

Software Pipelining Procedure

1. Write algorithm in C code & verify.
2. Write 'C6x Linear Assembly code.
3. Create dependency graph.
4. Allocate registers.
5. Create scheduling table.
6. Translate scheduling table to 'C6x code.

Write code in Linear Assembly (Step 2)

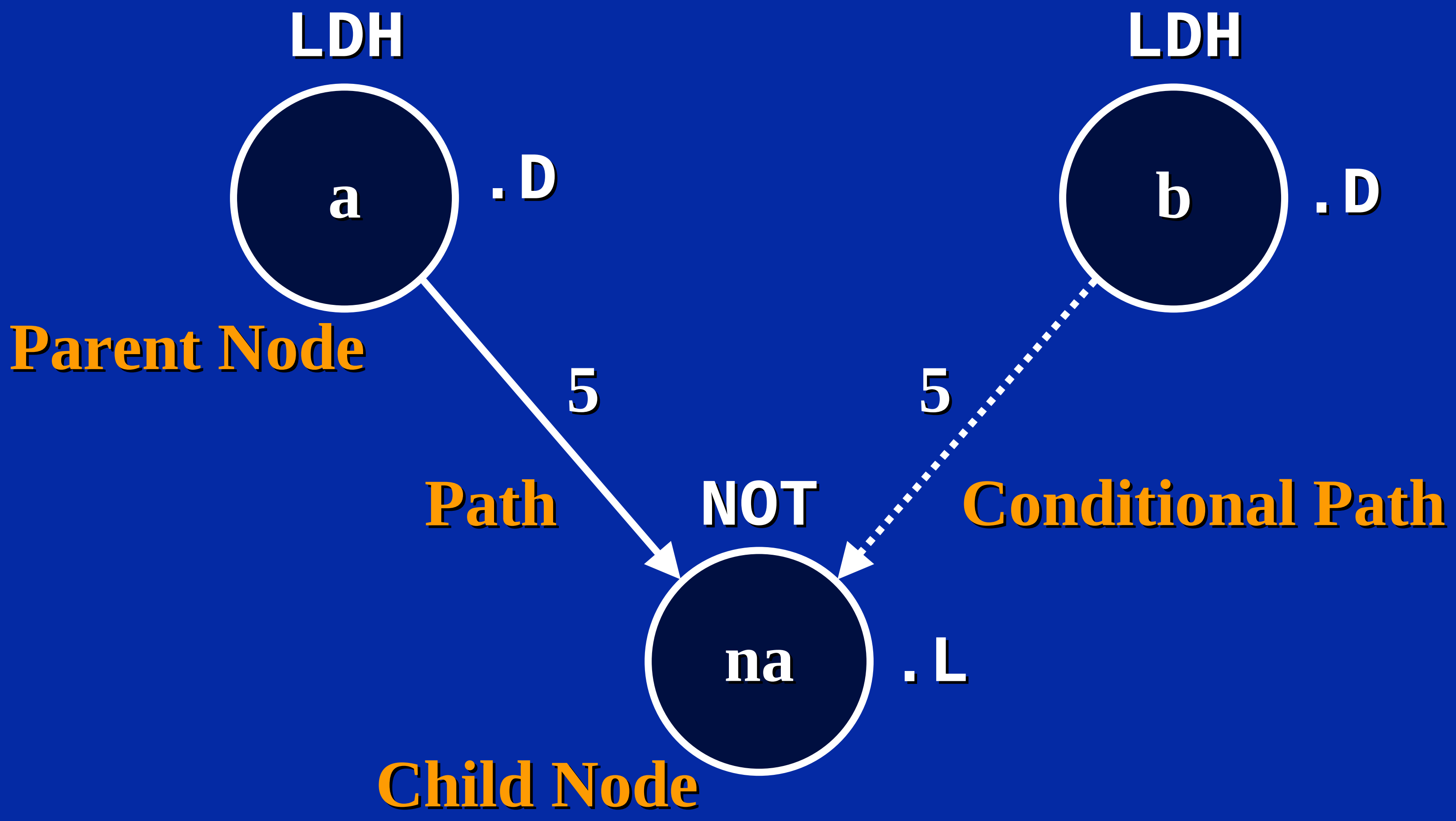
```
; for (i=0; i < count; i++)
; prod = m[i] * n[i];
; sum += prod;

loop:       ldh         *p_m++, m
            ldh         *p_n++, n
            mpy         m, n, prod
            add         prod, sum, sum

   [count]  sub         count, 1, count
   [count]  b           loop
```

1. No NOP's required.
2. No parallel instructions required.
3. You don't have to specify:
   - Functional units, or
   - Registers.

Software Pipelining Procedure

1. Write algorithm in C code & verify.
2. Write ‘C6x Linear Assembly code.
3. Create a dependency graph (4 steps).
4. Allocate registers.
5. Create scheduling table.
6. Translate scheduling table to ‘C6x code.

Dependency Graph Terminology



Dependency Graph Steps

(a) Draw the algorithm nodes and paths.

(b) Write the number of cycles it takes for each instruction to complete execution.

(c) Assign “required” function units to each node.

(d) Partition the nodes to A and B sides and assign sides to all functional units.

Dependency Graph (Step a)

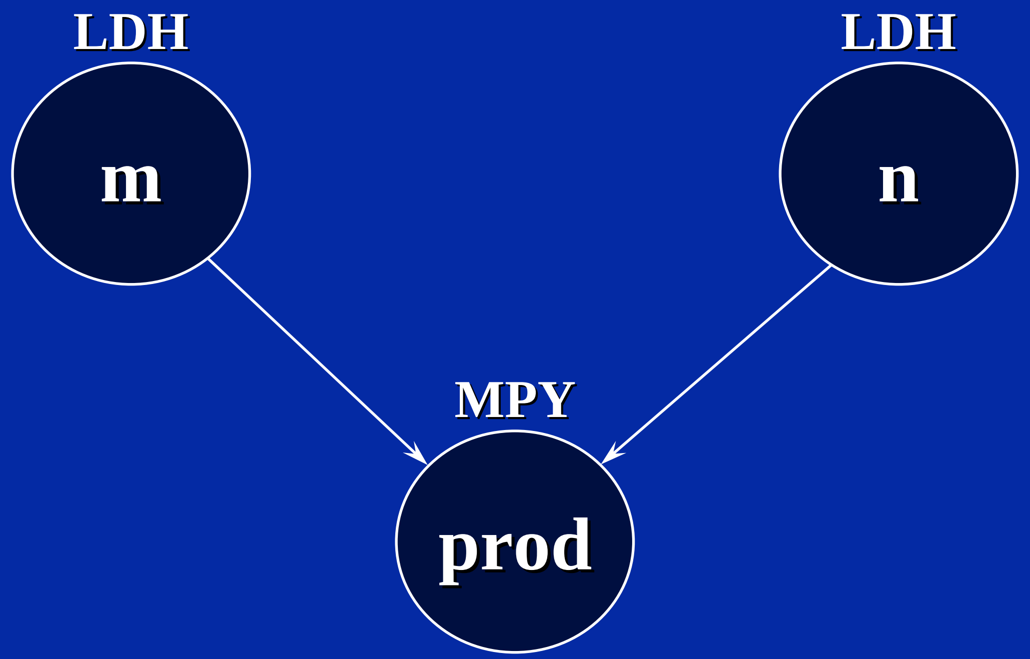
- In this step each instruction is represented by a node.
- The node is represented by a circle, where:
  - Outside: write instruction.
  - Inside: register where result is written.
- Nodes are then connected by paths showing the data flow.

Note: Conditional paths are represented by dashed lines.

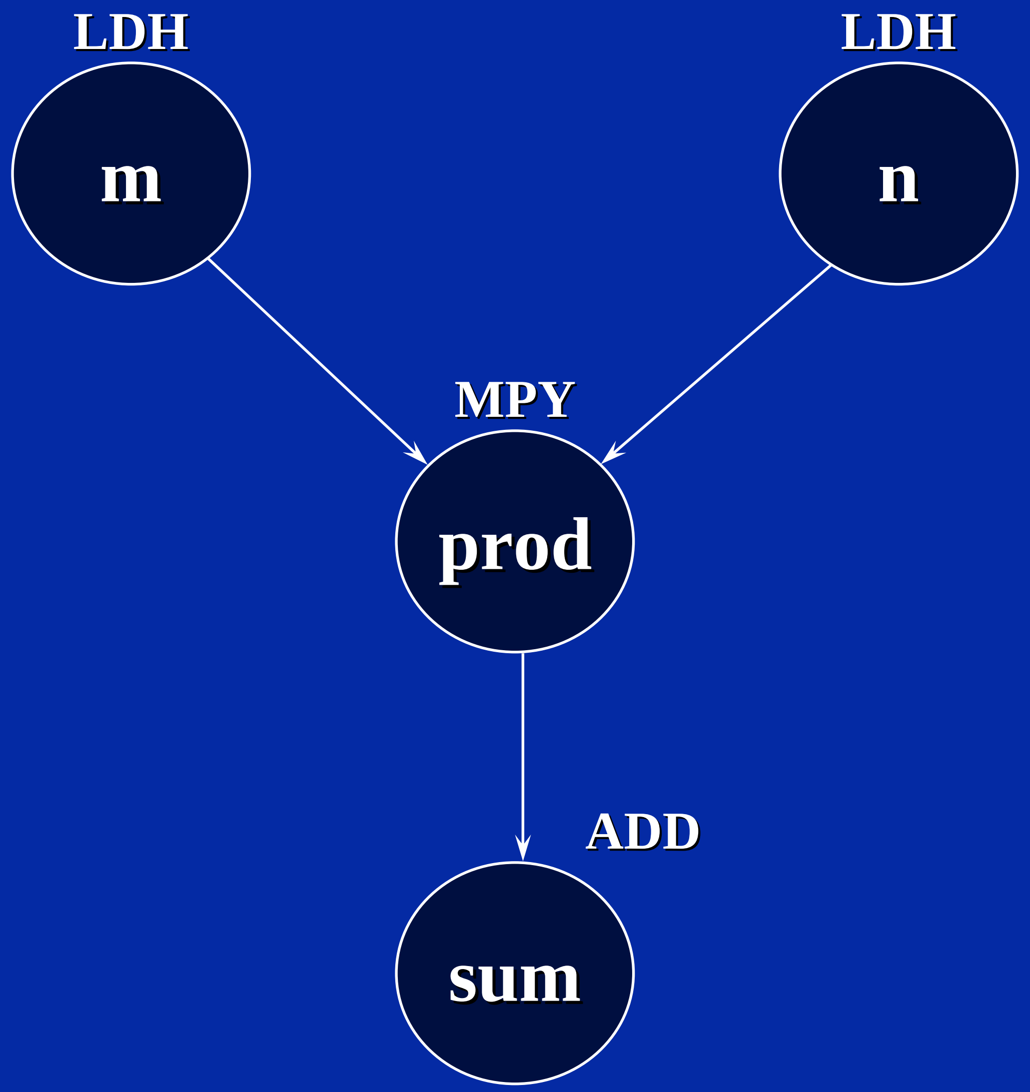
Dependency Graph (Step a)

Dependency Graph (Step a)

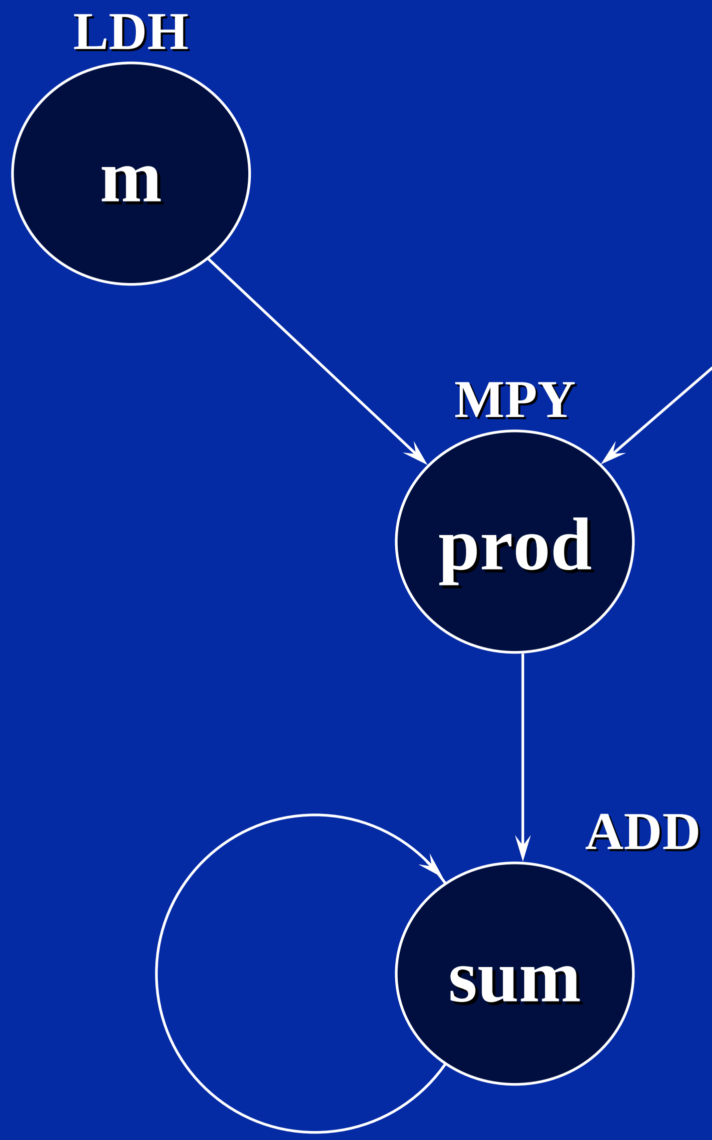
Dependency Graph (Step a)



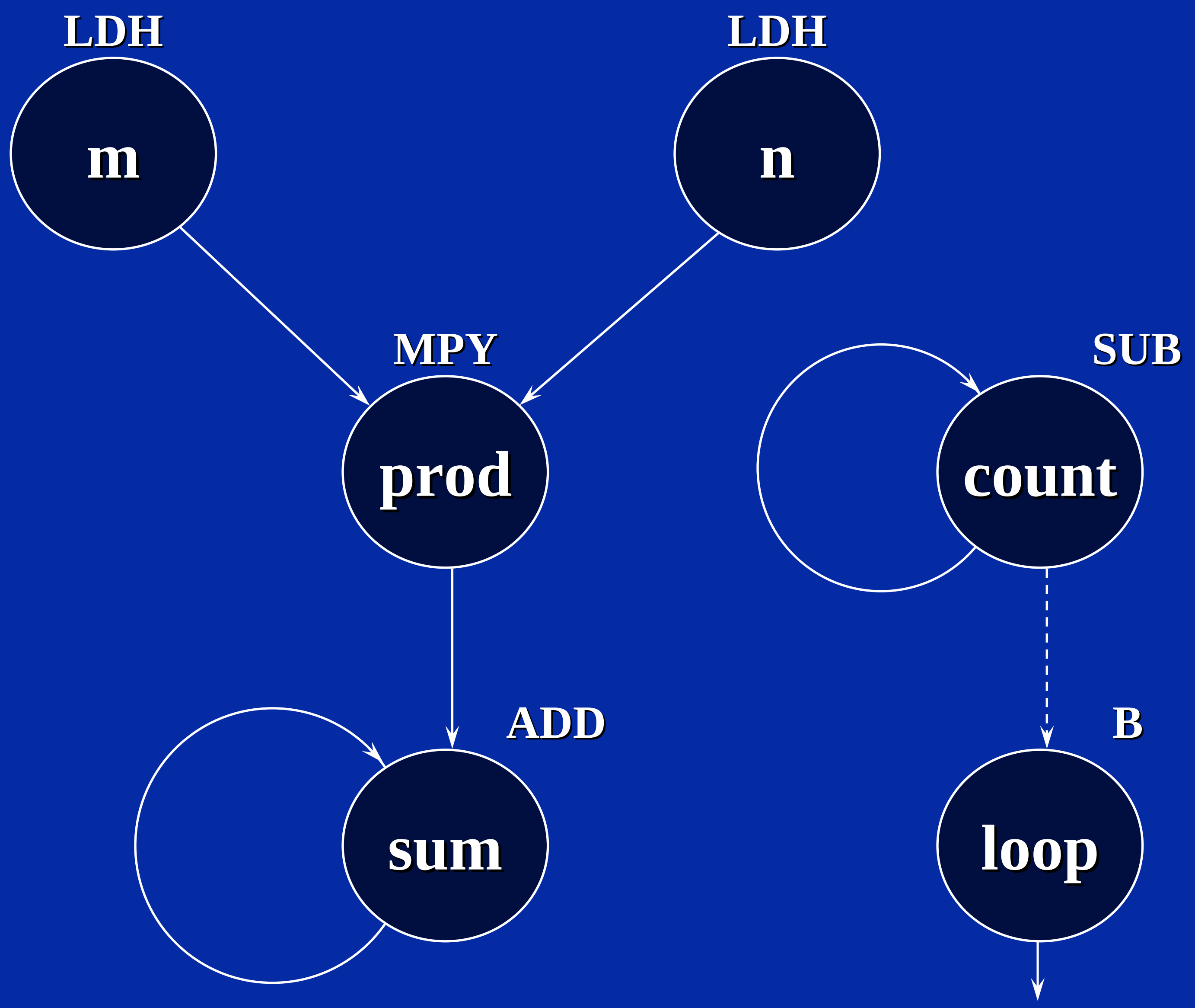
Dependency Graph (Step a)



Dependency Graph (Step a)



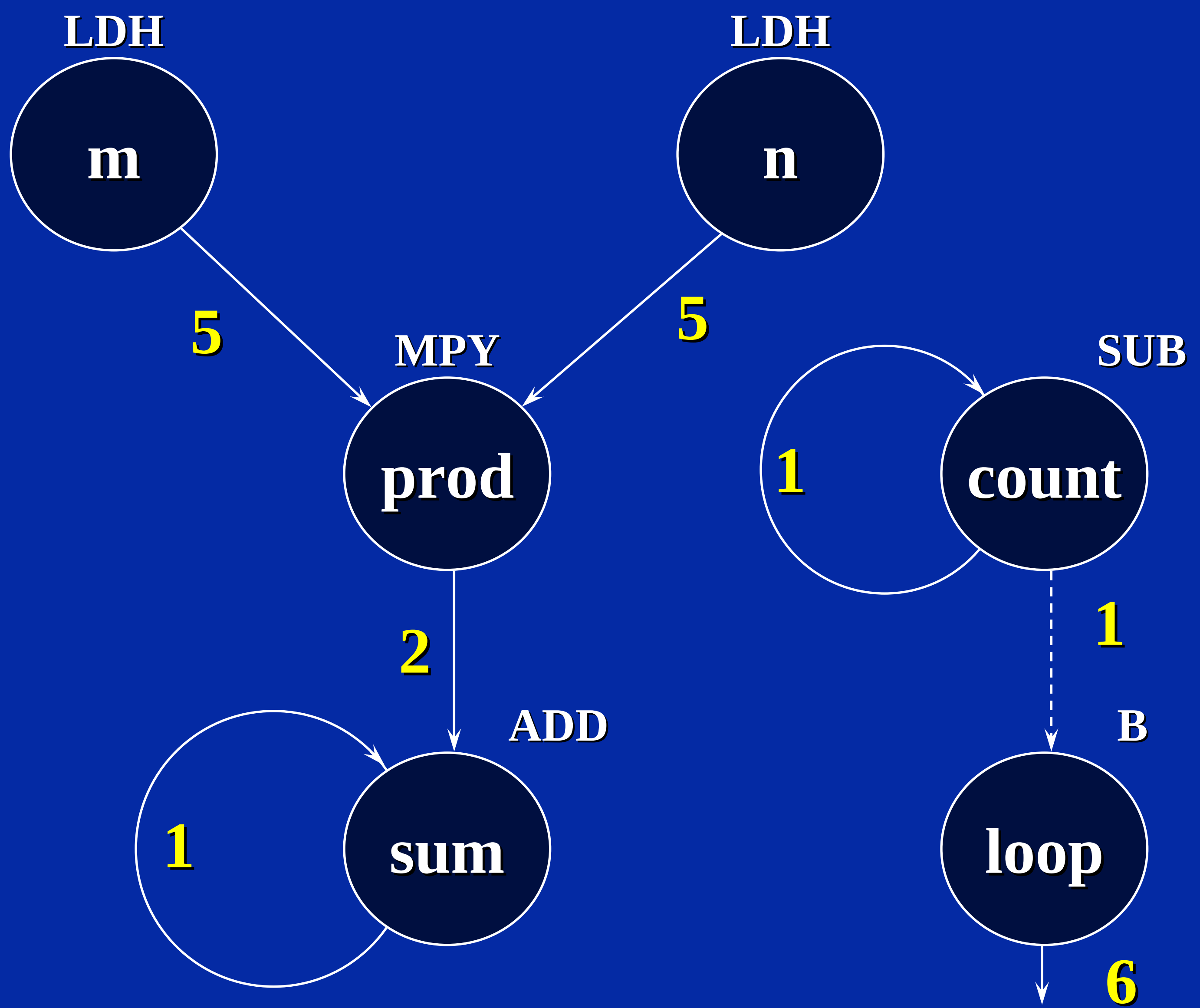
Dependency Graph (Step a)



Dependency Graph (Step b)

- In this step the number of cycles it takes for each instruction to complete execution is added to the dependency graph.
- It is written along the associated data path.

Dependency Graph (Step b)

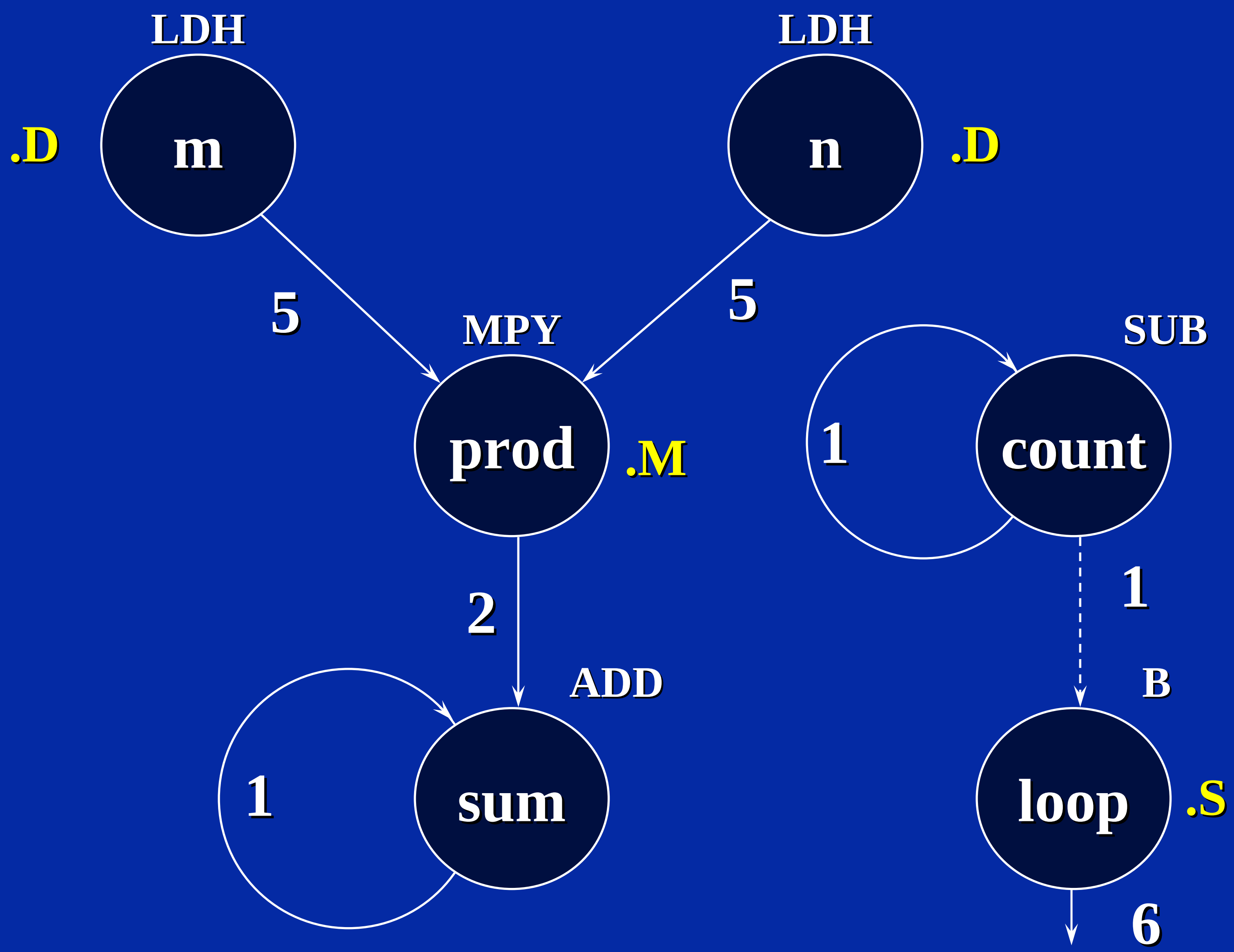


Dependency Graph (Step c)

- In this step functional units are assigned to each node.
- It is advantageous to start allocating units to instructions which require a specific unit:
  - Load/Store.
  - Branch.
- We do not need to be concerned with multiply as this is the only operation that the .M unit performs.

Note: The side is not allocated at this stage.

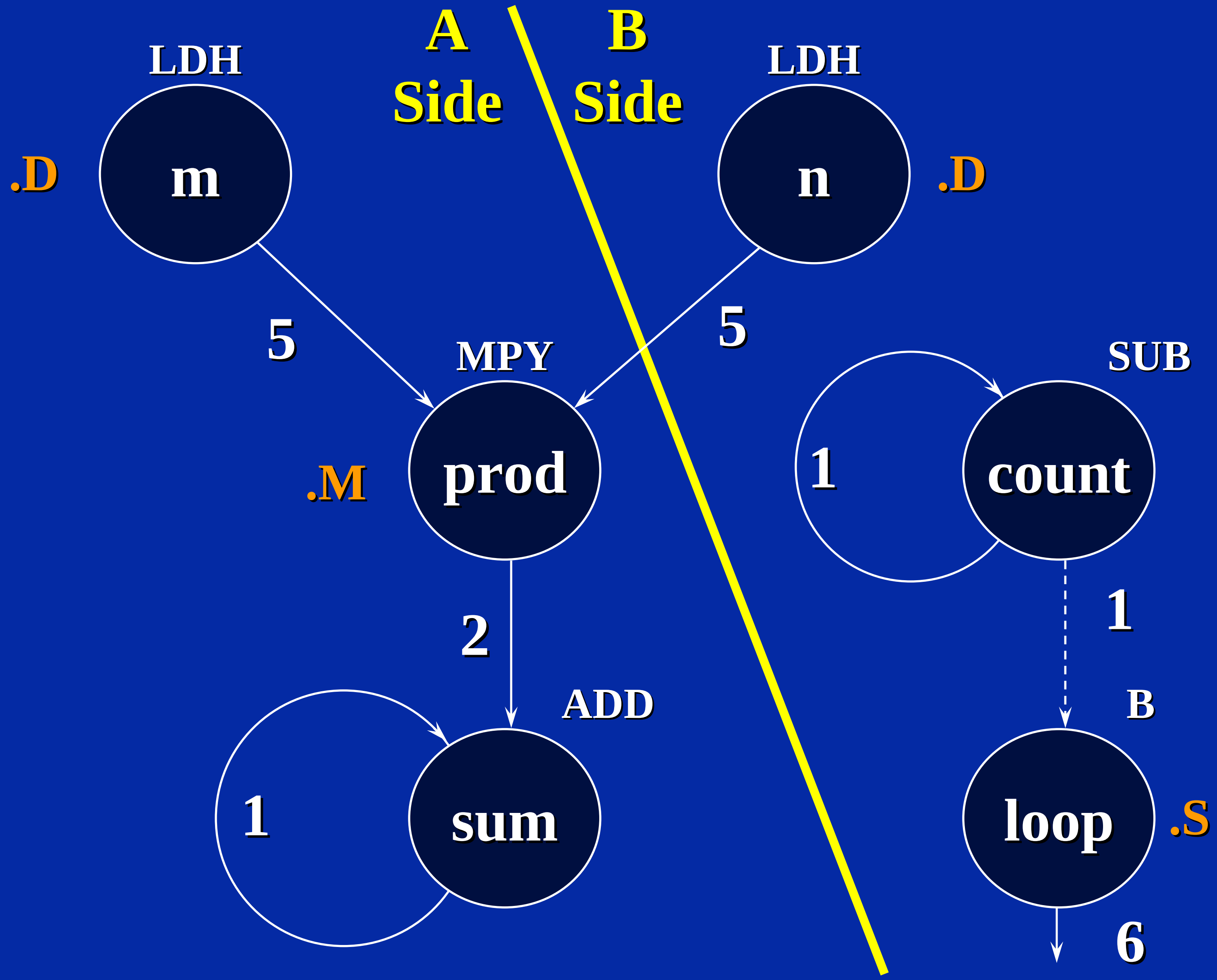
Dependency Graph (Step c)



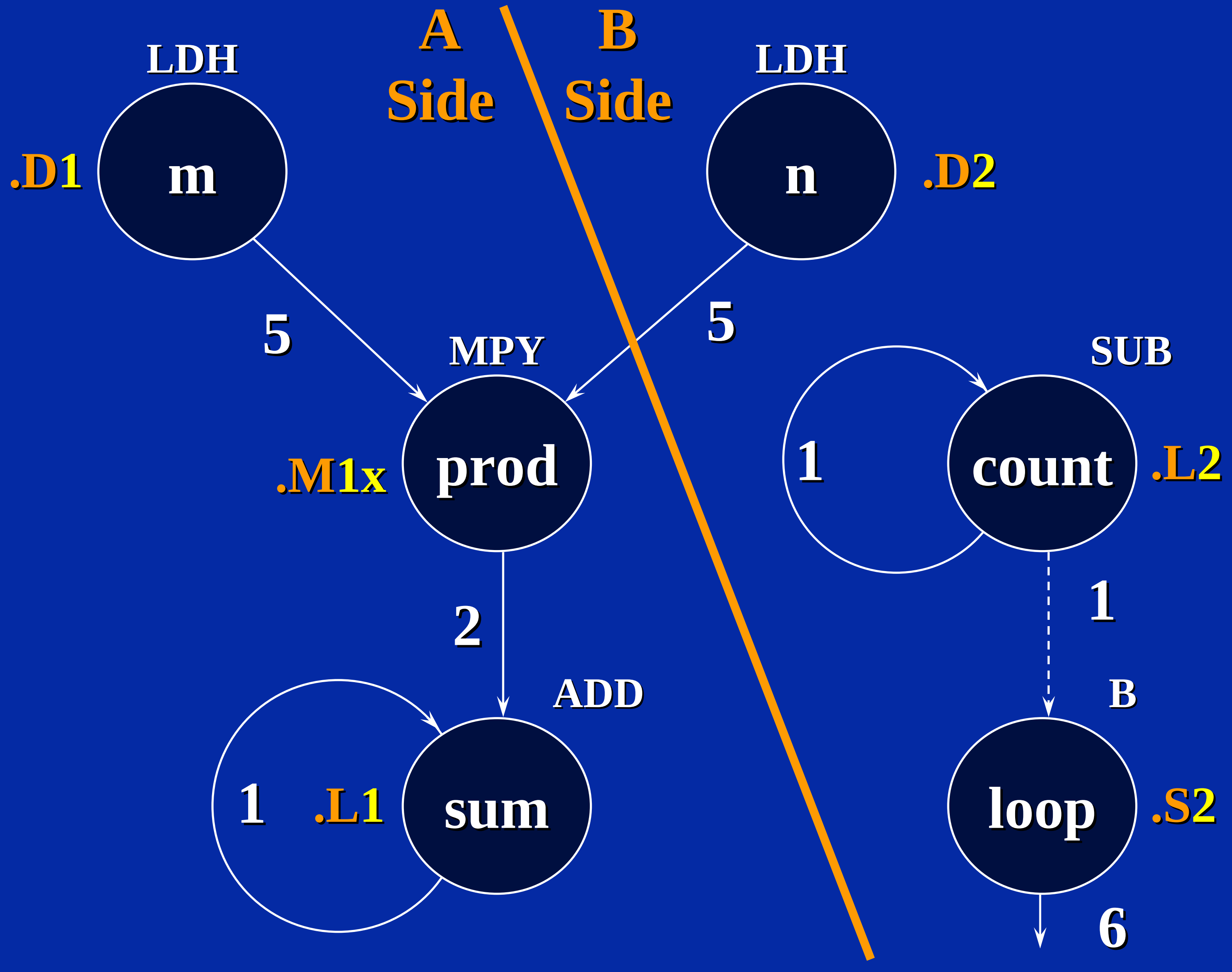
Dependency Graph (Step d)

- The data path is partitioned into side A and B at this stage.
- To optimise code we need to ensure that a maximum number of units are used with a minimum number of cross paths.
- To make the partition visible on the dependency graph a line is used.
- The side can then be added to the functional units associated with each instruction or node.

Dependency Graph (Step d)



Dependency Graph (Step d)



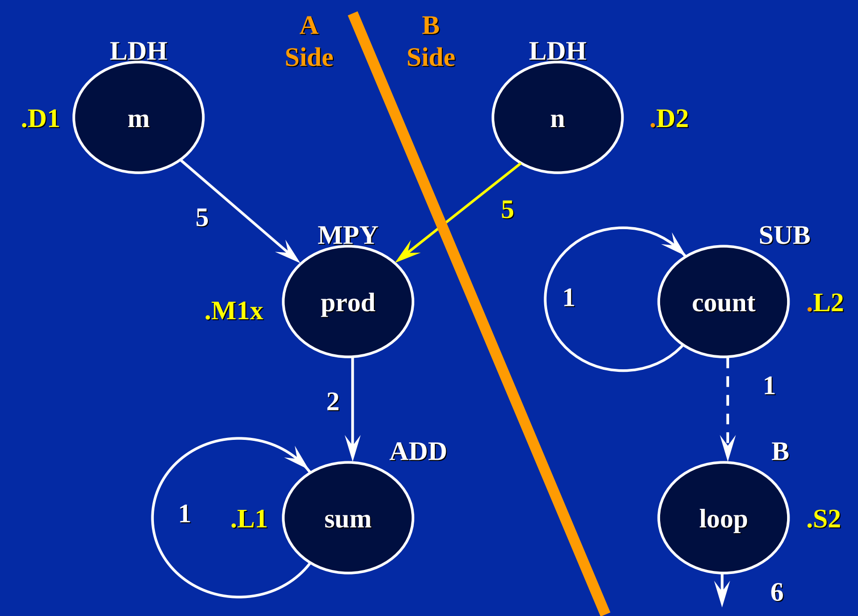
Software Pipelining Procedure

1. Write algorithm in C code & verify.
2. Write ‘C6x Linear Assembly code.
3. Create a dependency graph (4 steps).
4. Allocate registers.
5. Create scheduling table.
6. Translate scheduling table to ‘C6x code.

Step 4 - Allocate Functional Units

✓	.L1	sum
✓	.M1	prod
✓	.D1	m
	.S1	
	x1	.M1x
✓	.L2	count
✓	.M2	
	.D2	n
✓	.S2	loop
✓	x2	

Do we have enough functional units to code this algorithm in a single-cycle loop?



Step 4 - Allocate Registers

Content of Register File A	Reg. A	Reg. B	Content of Register File B
	A0	B0	count
&a	A1	B1	&b
a	A2	B2	b
prod	A3	B3	
sum	A4	B4	
	...	...	
	A15	B15	

Software Pipelining Procedure

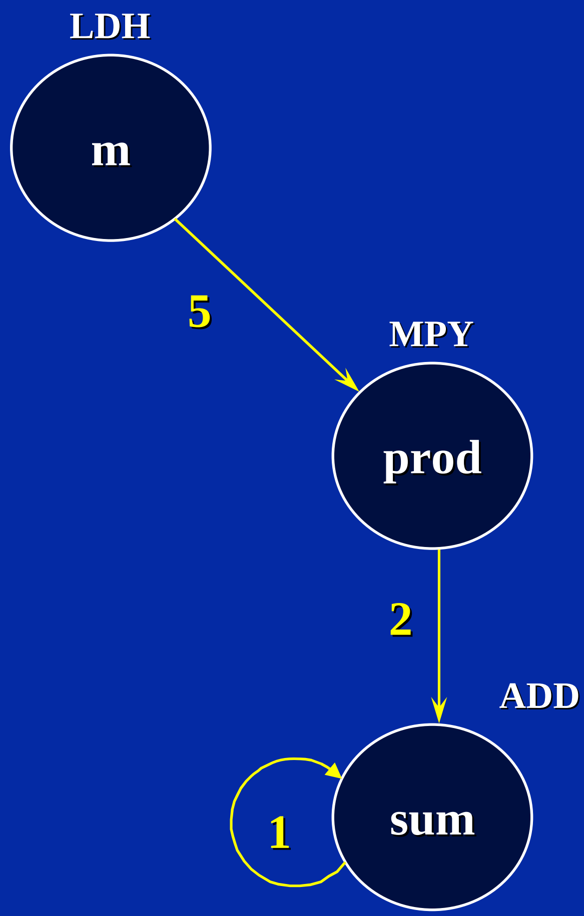
1. Write algorithm in C code & verify.
2. Write ‘C6x Linear Assembly code.
3. Create a dependency graph (4 steps).
4. Allocate registers.
5. Create scheduling table.
6. Translate scheduling table to ‘C6x code.

Step 5 - Create Scheduling Table

	PROLOG							LOOP
	1	2	3	4	5	6	7	8
.L1								
.L2								
.S1								
.S2								
.M1								
.M2								
.D1								
.D2								

How do we know the loop ends up in cycle 8?

Length of Prolog



Answer:

- Count up the length of longest path, in this case we have:

  5 + 2 + 1 = 8 cycles

Scheduling Table

	PROLOG							LOOP
	1	2	3	4	5	6	7	8
.L1								
.L2								
.S1								
.S2								
.M1								
.M2								
.D1								
.D2								

Scheduling Table

	PROLOG							LOOP
	1	2	3	4	5	6	7	8
.L1								add
.L2								
.S1								
.S2			B	*	*	*	*	*
.M1						mpy	*	*
.M2								
.D1	ldh a	*	*	*	*	*	*	*
.D2	ldh b	*	*	*	*	*	*	*

Where do we want to branch?

Branch here

Scheduling Table

	PROLOG							LOOP
	1	2	3	4	5	6	7	8
.L1								add
.L2		sub	*	*	*	*	*	*
.S1								
.S2			B	*	*	*	*	*
.M1						mpy	*	*
.M2								
.D1	ldh m	*	*	*	*	*	*	*
.D2	ldh n	*	*	*	*	*	*	*

Software Pipelining Procedure

1. Write algorithm in C code & verify.
2. Write ‘C6x Linear Assembly code.
3. Create a dependency graph (4 steps).
4. Allocate registers.
5. Create scheduling table.
6. Translate scheduling table to ‘C6x code.

Translate Scheduling Table to 'C6x Code

```
C1          ldh  .D1   *A1++,A2
 ||         ldh  .D2   *B1++,B2
```

	1
.L1	
.L2	
.S1	
.S2	
.M1	
.M2	
.D1	ldh m
.D2	ldh n

LOOP
7
add
*
*
*
*
*

Translate Scheduling Table to 'C6x Code

	1	2
.L1		
.L2		sub
.S1		
.S2		
.M1		
.M2		
.D1	ldh a	*
.D2	ldh b	*

```
C1          ldh  .D1   *A1++,A2
 ||         ldh  .D2   *B1++,B2

C2          ldh  .D1   *A1++,A2
 ||         ldh  .D2   *B1++,B2
 ||  [B0]   sub  .L2   B0,1,B0
```

Translate Scheduling Table to 'C6x Code

	1	2	3
.L1			
.L2		sub	*
.S1			
.S2			B
.M1			
.M2			
.D1	ldh m	*	*
.D2	ldh n	*	*

```
C1          ldh  .D1   *A1++,A2
 ||         ldh  .D2   *B1++,B2

C2          ldh  .D1   *A1++,A2
 ||         ldh  .D2   *B1++,B2
 || [B0]    sub  .L2   B0,1,B0

C3          ldh  .D1   *A1++,A2
 ||         ldh  .D2   *B1++,B2
 || [B0]    sub  .L2   B0,1,B0
 || [B0]    B    .S2   loop
```

Translate Scheduling Table to 'C6x Code

	1	2	3	4
.L1				
.L2		sub	*	*
.S1				
.S2			B	*
.M1				
.M2				
.D1	ldh m	*	*	*
.D2	ldh n	*	*	*

```
C1        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2

C2        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2
 || [B0] sub .L2  B0,1,B0

C3        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2
 || [B0] sub .L2  B0,1,B0
 || [B0] B   .S2  loop

C4        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2
 || [B0] sub .L2  B0,1,B0
 || [B0] B   .S2  loop
```

Translate Scheduling Table to 'C6x Code

```
C1        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2

C2        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2
 || [B0]  sub .L2  B0,1,B0

C3        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2
 || [B0]  sub .L2  B0,1,B0
 || [B0]  B   .S2  loop

C4        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2
 || [B0]  sub .L2  B0,1,B0
 || [B0]  B   .S2  loop

C5        ldh .D1  *A1++,A2
 ||       ldh .D2  *B1++,B2
 || [B0]  sub .L2  B0,1,B0
 || [B0]  B   .S2  loop
```

	5	6	7	LOOP 8
.L1				add
.L2	sub	*	*	*
.S1				
.S2	B	*	*	*
.M1		mpy	*	*
.M2				
.D1	ldh	*	*	*
.D2	ldh	*	*	*

Translate Scheduling Table to 'C6x Code

```
C6          ldh .D1   *A1++,A2
  ||        ldh .D2   *B1++,B2
  || [B0]  sub .L2   B0,1,B0
  || [B0]  B   .S2   loop
  ||        mpy .M1x A2,B2,A3
```

	6	7	8 (LOOP)
.L1			add
.L2	sub	*	*
.S1			
.S2	B	*	*
.M1	mpy	*	*
.M2			
.D1	ldh	*	*
.D2	ldh	*	*

Translate Scheduling Table to 'C6x Code

```
C7          ldh .D1   *A1++,A2
  ||        ldh .D2   *B1++,B2
  || [B0]  sub .L2   B0,1,B0
  || [B0]  B   .S2   loop
  ||        mpy .M1x A2,B2,A3
```

	6	7	8 (LOOP)
.L1			add
.L2	sub	*	*
.S1			
.S2	B	*	*
.M1	mpy	*	*
.M2			
.D1	ldh	*	*
.D2	ldh	*	*

Translate Scheduling Table to 'C6x Code

```
*  Single-Cycle Loop

loop:      ldh .D1  *A1++,A2
 ||        ldh .D2  *B1++,B2
 || [B0] sub .L2  B0,1,B0
 || [B0] B   .S2  loop
 ||        mpy .M1x A2,B2,A3
 ||        add .L1  A4,A3,A4
```

	6	7	8 (LOOP)
.L1			add
.L2	sub	*	*
.S1			
.S2	B	*	*
.M1	mpy	*	*
.M2			
.D1	ldh	*	*
.D2	ldh	*	*

Complete code

Translate Scheduling Table to 'C6x Code

- With this method we have only created the **prolog** and the **loop**.
- Therefore if the filter has a 100 taps, then we need to repeat the loop 100 times as we need 100 adds.
- This means that we are performing 107 loads. These **7 extra loads** may lead to **some illegal memory acesses**.

	PROLOG							LOOP
	1	2	3	4	5	6	7	8
.L1								add
.L2		sub	sub	sub	sub	sub	sub	sub
.S1								
.S2			B	B	B	B	B	B
.M1						mpy	mpy	mpy
.M2								
.D1	ldh m	ldh m	ldh m	ldh m	ldh m	ldh m	ldh m	ldh m
.D2	ldh n	ldh n	ldh n	ldh n	ldh n	ldh n	ldh n	ldh n

Solution: The Epilog

We only created the Prolog and Loop …
What about the Epilog?

The Epilog can be extracted from your results as described below.

See example in the next slide.

Dot-Product with Epilog

Prolog

```
p1: ldh||ldh

p2: ldh||ldh
 || []sub

p3: ldh||ldh
 || []sub
 || []b

p4: ldh||ldh
 || []sub
 || []b

p5: ldh||ldh
 || []sub
 || []b

p6: ldh||ldh
 || mpy
 || []sub
 || []b

p7: ldh||ldh
 || mpy
 || []sub
 || []b
```

Loop

```
loop:     ldh
       || ldh
       || mpy
       || add
       || [] sub
       || [] b
```

Epilog

```
e1: mpy
 || add
```

Epilog = Loop - Prolog

And there is no sub or b in the epilog

Dot-Product with Epilog

Prolog

```
p1: ldh||ldh

p2: ldh||ldh
 || []sub

p3: ldh||ldh
 || []sub
 || []b

p4: ldh||ldh
 || []sub
 || []b

p5: ldh||ldh
 || []sub
 || []b

p6: ldh||ldh
 || mpy
 || []sub
 || []b

p7: ldh||ldh
 || mpy
 || []sub
 || []b
```

Loop

```
loop:     ldh
       || ldh
       || mpy
       || add
       || [] sub
       || [] b
```

Epilog

```
e1: mpy
 || add

e2: mpy
 || add
```

Epilog = Loop - Prolog

And there is no sub or b in the epilog

Dot-Product with Epilog

Prolog

```
p1: ldh||ldh

p2: ldh||ldh
 || []sub

p3: ldh||ldh
 || []sub
 || []b

p4: ldh||ldh
 || []sub
 || []b

p5: ldh||ldh
 || []sub
 || []b

p6: ldh||ldh
 || mpy
 || []sub
 || []b

p7: ldh||ldh
 || mpy
 || []sub
 || []b
```

Loop

```
loop:   ldh
     || ldh
     || mpy
     || add
     || [] sub
     || [] b
```

Epilog

```
e1: mpy
 || add

e2: mpy
 || add

e3: mpy
 || add
```

Epilog = Loop - Prolog

And there is no sub or b in the epilog

Dot-Product with Epilog

Prolog

```
p1: ldh||ldh

p2: ldh||ldh
 || []sub

p3: ldh||ldh
 || []sub
 || []b

p4: ldh||ldh
 || []sub
 || []b

p5: ldh||ldh
 || []sub
 || []b

p6: ldh||ldh
 || mpy
 || []sub
 || []b

p7: ldh||ldh
 || mpy
 || []sub
 || []b
```

Loop

```
loop:    ldh
      || ldh
      || mpy
      || add
      || [] sub
      || [] b
```

Epilog

```
e1: mpy
 || add

e2: mpy
 || add

e3: mpy
 || add

e4: mpy
 || add
```

Epilog = Loop - Prolog

And there is no sub or b in the epilog

Dot-Product with Epilog

Prolog

```
p1: ldh||ldh

p2: ldh||ldh
 || []sub

p3: ldh||ldh
 || []sub
 || []b

p4: ldh||ldh
 || []sub
 || []b

p5: ldh||ldh
 || []sub
 || []b

p6: ldh||ldh
 || mpy
 || []sub
 || []b

p7: ldh||ldh
 || mpy
 || []sub
 || []b
```

Loop

```
loop:     ldh
       || ldh
       || mpy
       || add
       || [] sub
       || [] b
```

Epilog

```
e1: mpy
 || add

e2: mpy
 || add

e3: mpy
 || add

e4: mpy
 || add

e5: mpy
 || add
```

Epilog = Loop - Prolog

And there is no sub or b in the epilog

Dot-Product with Epilog

Prolog

```
p1: ldh||ldh

p2: ldh||ldh
 || []sub

p3: ldh||ldh
 || []sub
 || []b

p4: ldh||ldh
 || []sub
 || []b

p5: ldh||ldh
 || []sub
 || []b

p6: ldh||ldh
 || mpy
 || []sub
 || []b

p7: ldh||ldh
 || mpy
 || []sub
 || []b
```

Loop

```
loop:    ldh
      || ldh
      || mpy
      || add
      || [] sub
      || [] b
```

Epilog

```
e1: mpy
 || add

e2: mpy
 || add

e3: mpy
 || add

e4: mpy
 || add

e5: mpy
 || add

e6: add
```

Epilog = Loop - Prolog

And there is no sub or b in the epilog

Dot-Product with Epilog

Prolog

```
p1: ldh||ldh

p2: ldh||ldh
 || []sub

p3: ldh||ldh
 || []sub
 || []b

p4: ldh||ldh
 || []sub
 || []b

p5: ldh||ldh
 || []sub
 || []b

p6: ldh||ldh
 || mpy
 || []sub
 || []b

p7: ldh||ldh
 || mpy
 || []sub
 || []b
```

Loop

```
loop:   ldh
     || ldh
     || mpy
     || add
     || [] sub
     || [] b
```

Epilog

```
e1: mpy
 || add

e2: mpy
 || add

e3: mpy
 || add

e4: mpy
 || add

e5: mpy
 || add

e6: add

e7: add
```

Epilog = Loop - Prolog

And there is no sub or b in the epilog

Scheduling Table: Prolog, Loop and Epilog

	Prologue							Loop	Epilogue						
Cycle / Unit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
.D1	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH							
.D2	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH							
.L1								ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD
.L2		SUB	SUB	SUB	SUB	SUB	SUB	SUB							
.S1															
.S2			B	B	B	B	B	B							
.M1						MPY	MPY	MPY	MPY	MPY	MPY	MPY	MPY		
.M2															