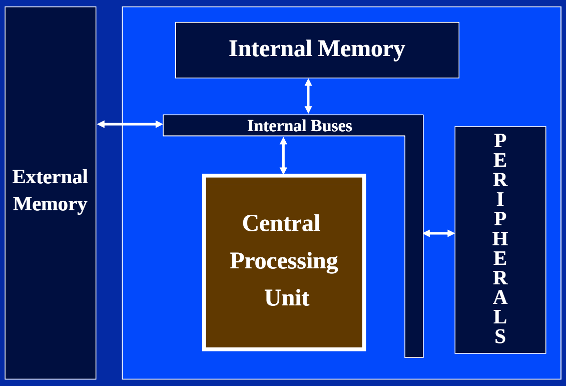
Chapter 2

TMS320C6000 Architectural Overview

Learning Objectives

- Describe C6000 CPU architecture.
- Introduce some basic instructions.
- Describe the C6000 memory map.
- Provide an overview of the peripherals.

General DSP System Block Diagram



Implementation of Sum of Products (SOP)

It has been shown in Chapter 1 that SOP is the key element for most DSP algorithms.

So let's write the code for this algorithm and at the same time discover the C6000 architecture.

$$Y = \sum_{n=1}^{N} a_n * x_n$$

$$= a_1 * x_1 + a_2 * x_2 + \ldots + a_N * x_N$$

Two basic operations are required for this algorithm.

(1) Multiplication

(2) Addition

Therefore two basic instructions are required

Implementation of Sum of Products (SOP)

So let's implement the SOP algorithm!

The implementation in this module will be done in assembly.

$$Y = \sum_{n=1}^{N} a_n * x_n$$

$$= a_1 * x_1 + a_2 * x_2 + \ldots + a_N * x_N$$

Two basic operations are required for this algorithm.

(1) Multiplication

(2) Addition

Therefore two basic instructions are required

Multiply (MPY)

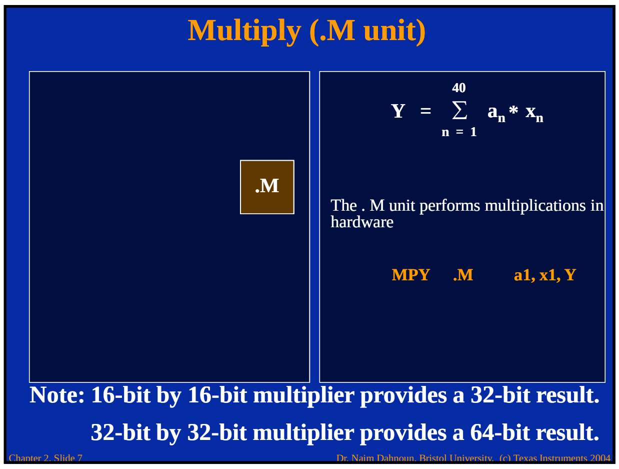
$$Y = \sum_{n=1}^{N} a_n * x_n$$

$$= a_1 * x_1 + a_2 * x_2 + \ldots + a_N * x_N$$

The multiplication of a_1 by x_1 is done in assembly by the following instruction:

```
MPY          a1, x1, Y
```

This instruction is performed by a multiplier unit that is called ".M"



Multiply (.M unit)

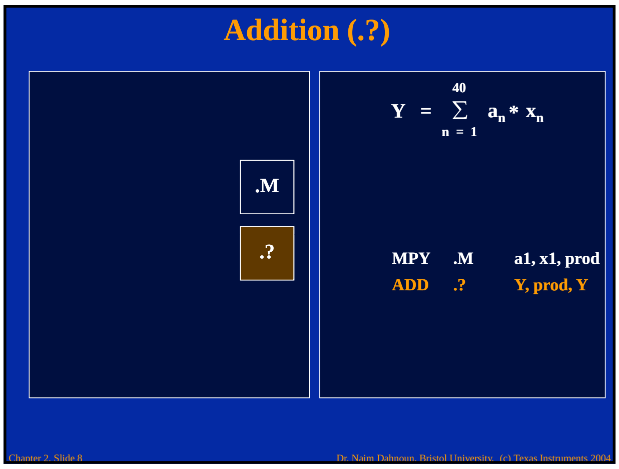
$$Y = \sum_{n=1}^{40} a_n * x_n$$

The . M unit performs multiplications in hardware

```
MPY    .M    a1, x1, Y
```

Note: 16-bit by 16-bit multiplier provides a 32-bit result.

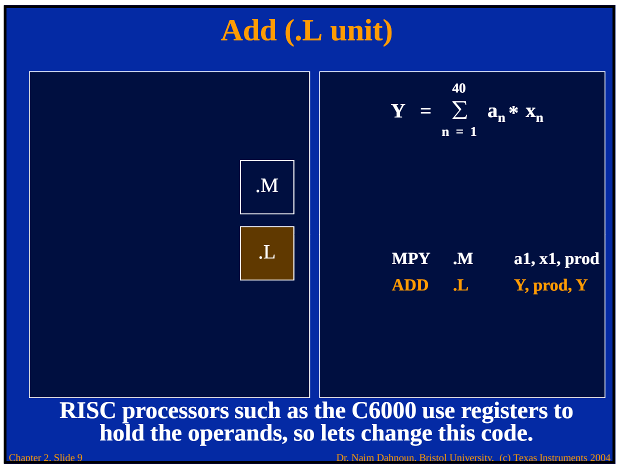
32-bit by 32-bit multiplier provides a 64-bit result.



Addition (.?)

$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
MPY    .M    a1, x1, prod
ADD    .?    Y, prod, Y
```

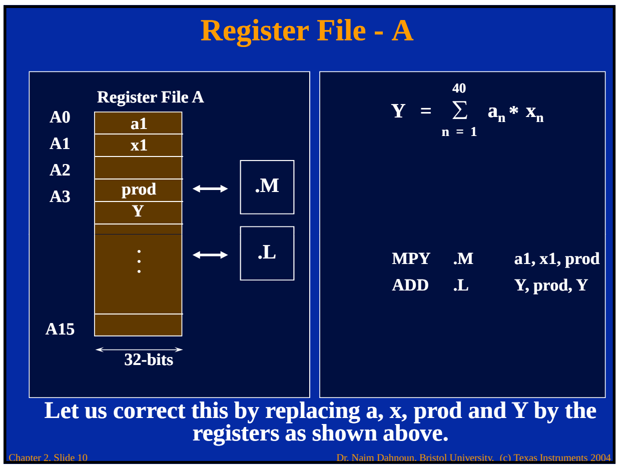


Add (.L unit)

$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
MPY    .M    a1, x1, prod
ADD    .L    Y, prod, Y
```

RISC processors such as the C6000 use registers to hold the operands, so lets change this code.

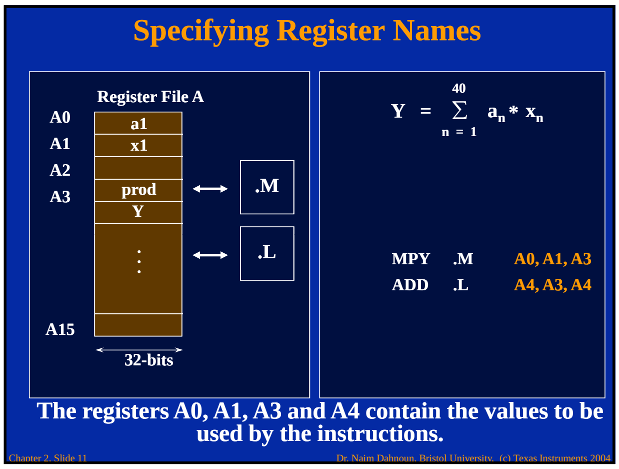


Register File - A

$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
MPY    .M    a1, x1, prod
ADD    .L    Y, prod, Y
```

Let us correct this by replacing a, x, prod and Y by the registers as shown above.

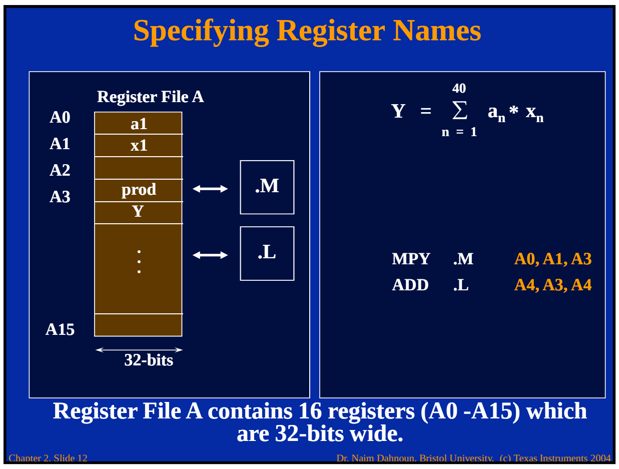


Specifying Register Names

$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
MPY    .M    A0, A1, A3
ADD    .L    A4, A3, A4
```

The registers A0, A1, A3 and A4 contain the values to be used by the instructions.

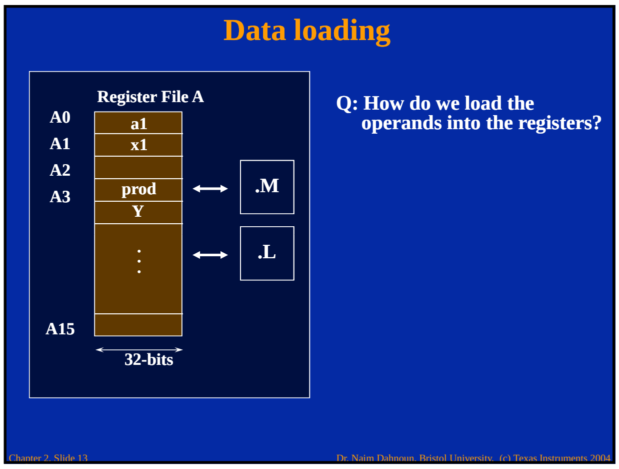


Specifying Register Names

$$Y = \sum_{n=1}^{40} a_n * x_n$$

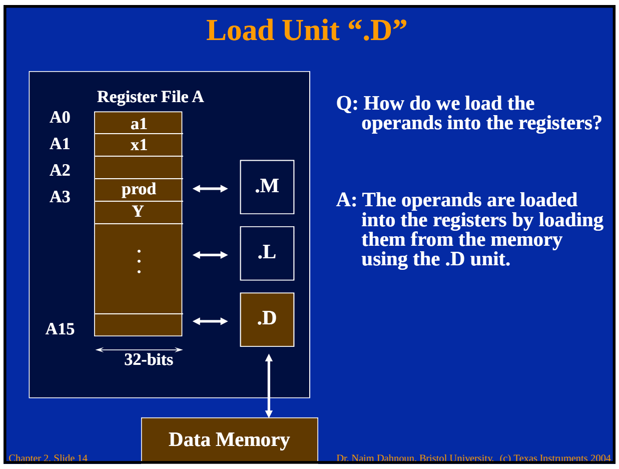
```
MPY    .M    A0, A1, A3
ADD    .L    A4, A3, A4
```

Register File A contains 16 registers (A0 -A15) which are 32-bits wide.



Data loading

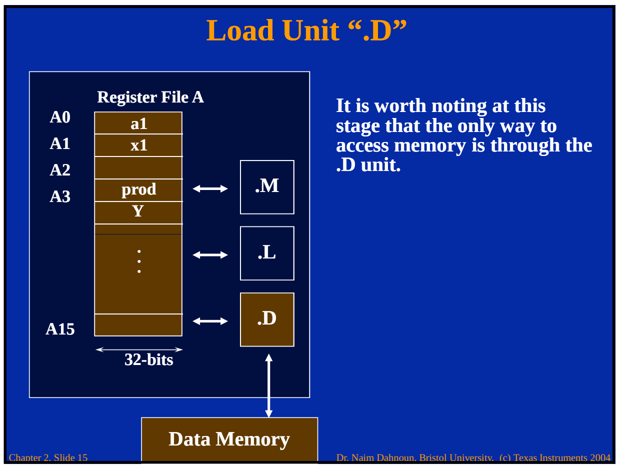
Q: How do we load the operands into the registers?



Load Unit ".D"

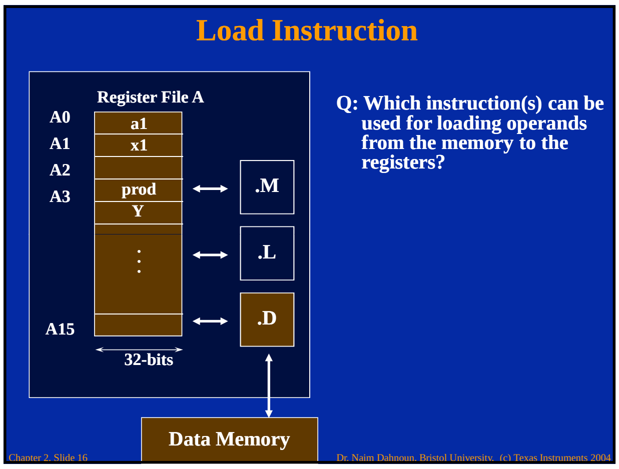
Q: How do we load the operands into the registers?

A: The operands are loaded into the registers by loading them from the memory using the .D unit.



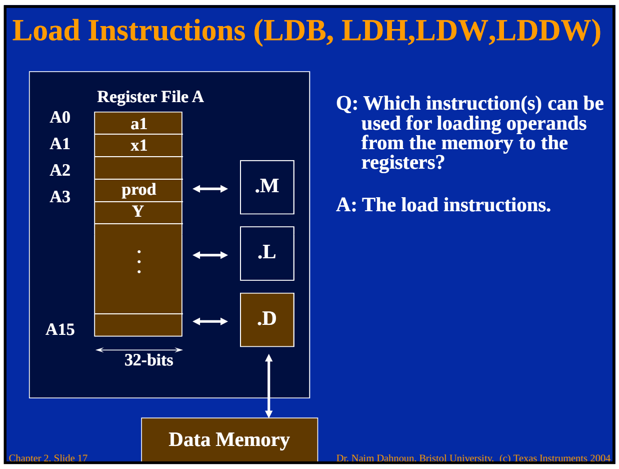
Load Unit ".D"

It is worth noting at this stage that the only way to access memory is through the .D unit.



Load Instruction

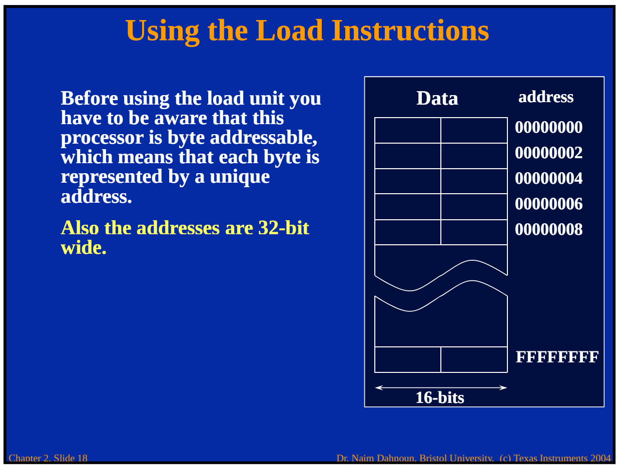
Q: Which instruction(s) can be used for loading operands from the memory to the registers?



Load Instructions (LDB, LDH,LDW,LDDW)

Q: Which instruction(s) can be used for loading operands from the memory to the registers?

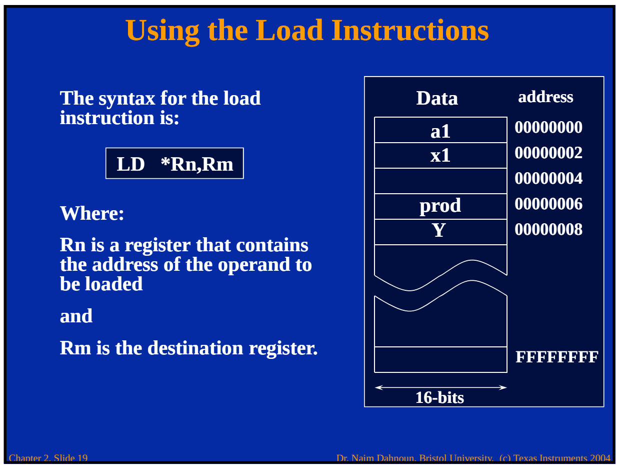
A: The load instructions.



Using the Load Instructions

Before using the load unit you have to be aware that this processor is byte addressable, which means that each byte is represented by a unique address.

Also the addresses are 32-bit wide.



Using the Load Instructions

The syntax for the load instruction is:

LD *Rn,Rm

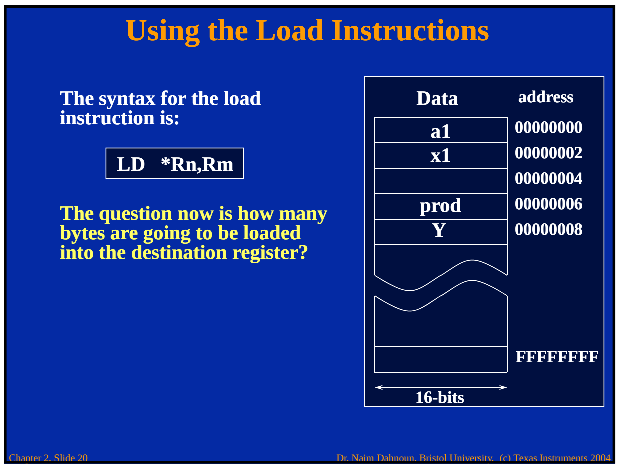
Where:

Rn is a register that contains the address of the operand to be loaded

and

Rm is the destination register.

Data	address
a1	00000000
x1	00000002
	00000004
prod	00000006
Y	00000008
	FFFFFFFF
16-bits	



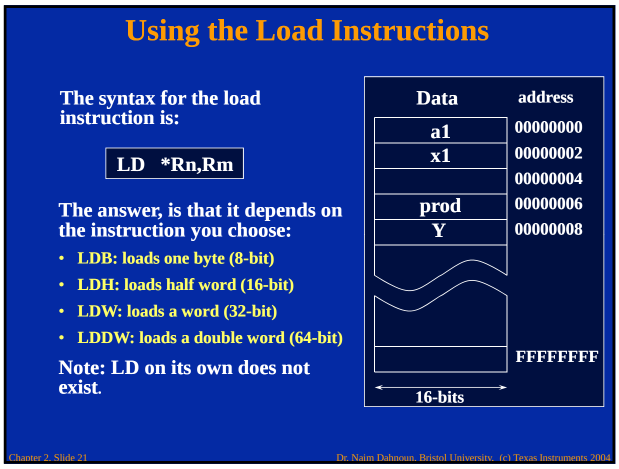
Using the Load Instructions

The syntax for the load instruction is:

LD *Rn,Rm

The question now is how many bytes are going to be loaded into the destination register?

Data	address
a1	00000000
x1	00000002
	00000004
prod	00000006
Y	00000008
	FFFFFFFF
16-bits	



Using the Load Instructions

The syntax for the load instruction is:

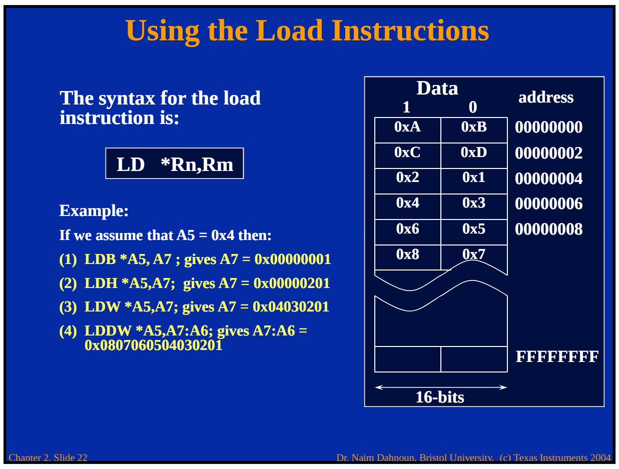
LD *Rn,Rm

The answer, is that it depends on the instruction you choose:

- LDB: loads one byte (8-bit)
- LDH: loads half word (16-bit)
- LDW: loads a word (32-bit)
- LDDW: loads a double word (64-bit)

Note: LD on its own does not exist.

Data	address
a1	00000000
x1	00000002
	00000004
prod	00000006
Y	00000008
	FFFFFFFF
16-bits	



Using the Load Instructions

The syntax for the load instruction is:

LD *Rn,Rm

Example:

If we assume that A5 = 0x4 then:

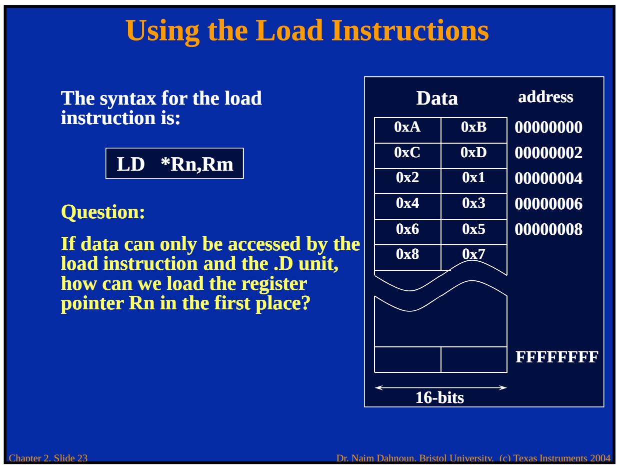
(1) LDB *A5, A7 ; gives A7 = 0x00000001

(2) LDH *A5,A7; gives A7 = 0x00000201

(3) LDW *A5,A7; gives A7 = 0x04030201

(4) LDDW *A5,A7:A6; gives A7:A6 = 0x0807060504030201

Data 1	Data 0	address
0xA	0xB	00000000
0xC	0xD	00000002
0x2	0x1	00000004
0x4	0x3	00000006
0x6	0x5	00000008
0x8	0x7	
		FFFFFFFF
16-bits		



Using the Load Instructions

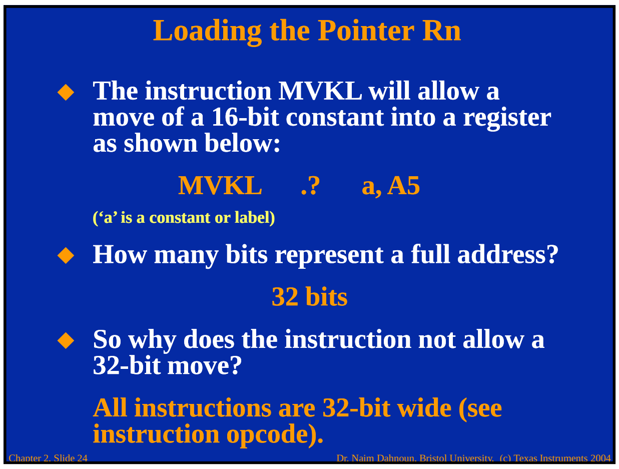
The syntax for the load instruction is:

LD *Rn,Rm

Question:

If data can only be accessed by the load instruction and the .D unit, how can we load the register pointer Rn in the first place?

Data		address
0xA	0xB	00000000
0xC	0xD	00000002
0x2	0x1	00000004
0x4	0x3	00000006
0x6	0x5	00000008
0x8	0x7	
		FFFFFFFF
16-bits		



Loading the Pointer Rn

- The instruction MVKL will allow a move of a 16-bit constant into a register as shown below:

  MVKL .? a, A5

  ('a' is a constant or label)

- How many bits represent a full address?

  32 bits

- So why does the instruction not allow a 32-bit move?

  All instructions are 32-bit wide (see instruction opcode).

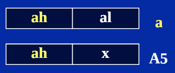
Loading the Pointer Rn

- To solve this problem another instruction is available:

MVKH

eg. MVKH .? a, A5

('a' is a constant or label)



- Finally, to move the 32-bit address to a register we can use:

```
MVKL    a, A5
MVKH    a, A5
```

Loading the Pointer Rn

- Always use MVKL then MVKH, look at the following examples:

Example 1

A5 = 0x87654321

```
MVKL        0x1234FABC, A5
A5 = 0xFFFFFABC  (sign extension)
```

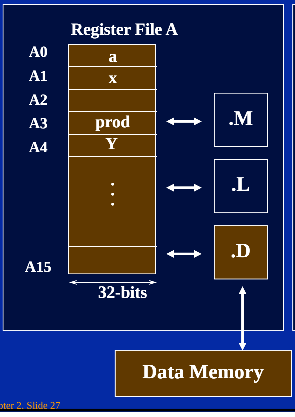
```
MVKH        0x1234FABC, A5
A5 = 0x1234FABC  ; OK
```

Example 2

```
MVKH        0x1234FABC, A5
A5 = 0x12344321
```

```
MVKL        0x1234FABC, A5
A5 = 0xFFFFFABC ; Wrong
```

LDH, MVKL and MVKH



```
MVKL          pt1, A5
MVKH          pt1, A5

MVKL          pt2, A6
MVKH          pt2, A6

LDH     .D    *A5, A0
LDH     .D    *A6, A1
MPY     .M    A0, A1, A3
ADD     .L    A4, A3, A4
```

pt1 and pt2 point to some locations in the data memory.

Creating a loop

So far we have only implemented the SOP for one tap only, i.e.

$$Y= a_1 * x_1$$

So let's create a loop so that we can implement the SOP for N Taps.

```
MVKL          pt1, A5
MVKH          pt1, A5

MVKL          pt2, A6
MVKH          pt2, A6

LDH     .D    *A5, A0
LDH     .D    *A6, A1
MPY     .M    A0, A1, A3
ADD     .L    A4, A3, A4
```

Creating a loop

So far we have only implemented the SOP for one tap only, i.e.

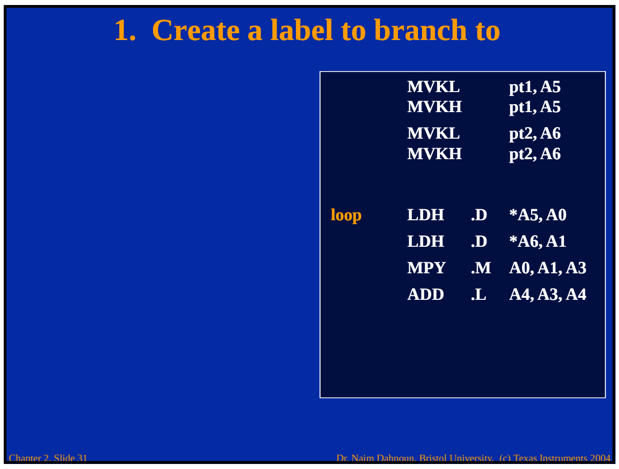
$$Y= a_1 * x_1$$

So let's create a loop so that we can implement the SOP for N Taps.

With the C6000 processors there are no dedicated instructions such as block repeat. The loop is created using the B instruction.

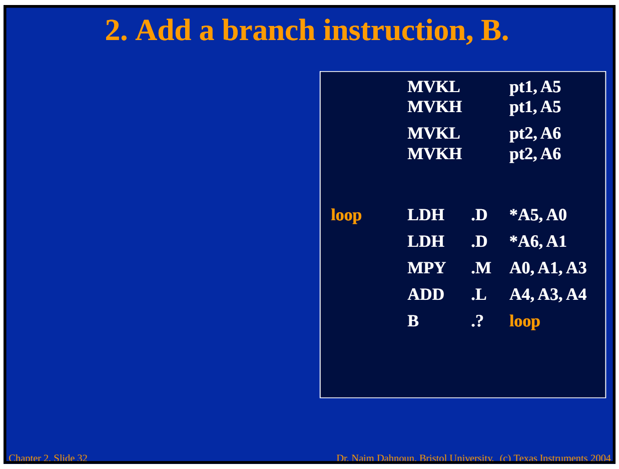
What are the steps for creating a loop

1. Create a label to branch to.
2. Add a branch instruction, B.
3. Create a loop counter.
4. Add an instruction to decrement the loop counter.
5. Make the branch conditional based on the value in the loop counter.



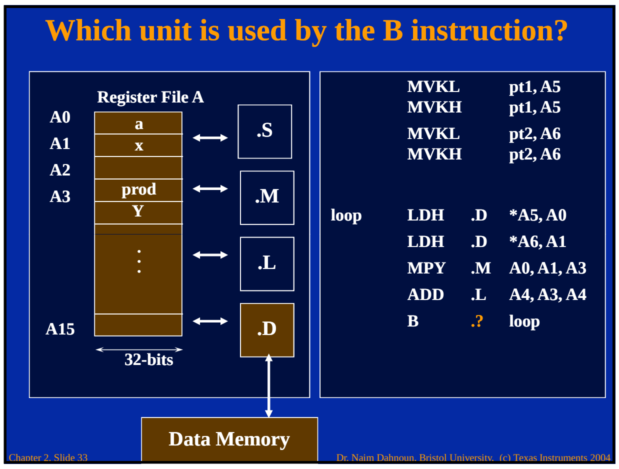
1. Create a label to branch to

```
        MVKL        pt1, A5
        MVKH        pt1, A5
        MVKL        pt2, A6
        MVKH        pt2, A6

loop    LDH    .D   *A5, A0
        LDH    .D   *A6, A1
        MPY    .M   A0, A1, A3
        ADD    .L   A4, A3, A4
```



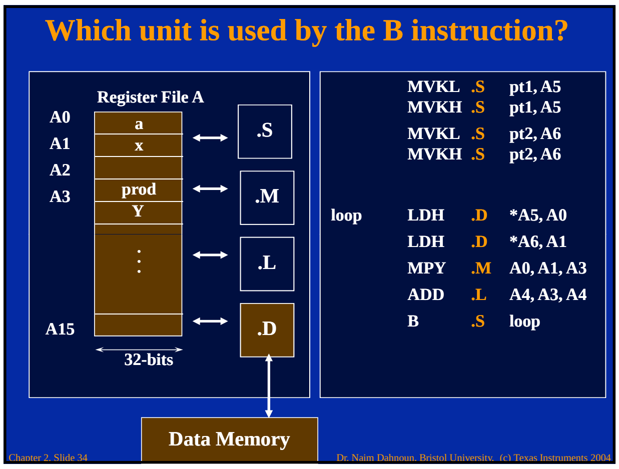
2. Add a branch instruction, B.

```
        MVKL        pt1, A5
        MVKH        pt1, A5
        MVKL        pt2, A6
        MVKH        pt2, A6

loop    LDH    .D   *A5, A0
        LDH    .D   *A6, A1
        MPY    .M   A0, A1, A3
        ADD    .L   A4, A3, A4
        B      .?   loop
```



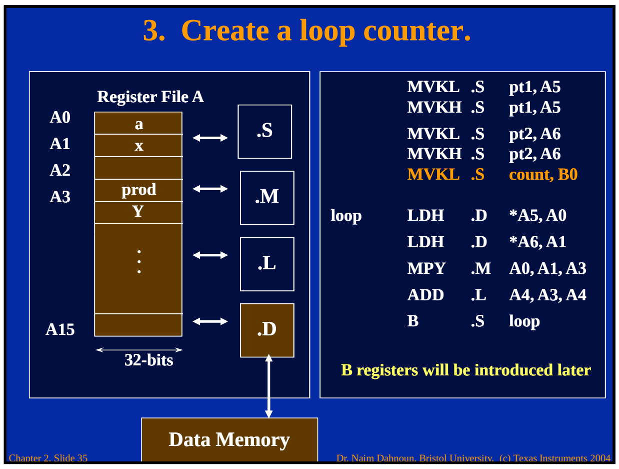
Which unit is used by the B instruction?

```
        MVKL        pt1, A5
        MVKH        pt1, A5
        MVKL        pt2, A6
        MVKH        pt2, A6

loop    LDH    .D   *A5, A0
        LDH    .D   *A6, A1
        MPY    .M   A0, A1, A3
        ADD    .L   A4, A3, A4
        B      .?   loop
```



Which unit is used by the B instruction?

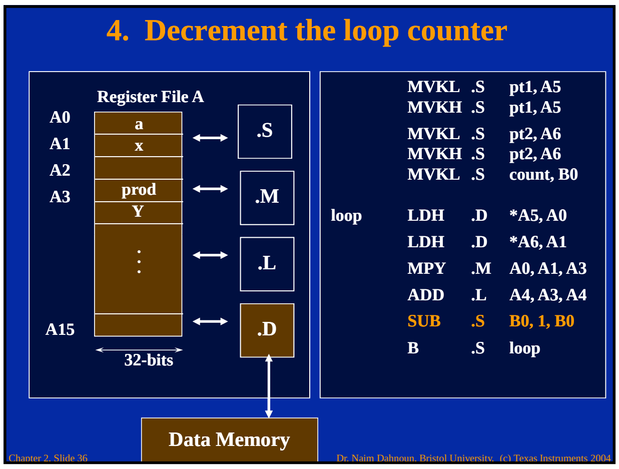
```
        MVKL   .S   pt1, A5
        MVKH   .S   pt1, A5
        MVKL   .S   pt2, A6
        MVKH   .S   pt2, A6

loop    LDH    .D   *A5, A0
        LDH    .D   *A6, A1
        MPY    .M   A0, A1, A3
        ADD    .L   A4, A3, A4
        B      .S   loop
```



3. Create a loop counter.

```
        MVKL   .S   pt1, A5
        MVKH   .S   pt1, A5
        MVKL   .S   pt2, A6
        MVKH   .S   pt2, A6
        MVKL   .S   count, B0

loop    LDH    .D   *A5, A0
        LDH    .D   *A6, A1
        MPY    .M   A0, A1, A3
        ADD    .L   A4, A3, A4
        B      .S   loop
```

B registers will be introduced later



4. Decrement the loop counter

```
        MVKL   .S   pt1, A5
        MVKH   .S   pt1, A5
        MVKL   .S   pt2, A6
        MVKH   .S   pt2, A6
        MVKL   .S   count, B0

loop    LDH    .D   *A5, A0
        LDH    .D   *A6, A1
        MPY    .M   A0, A1, A3
        ADD    .L   A4, A3, A4
        SUB    .S   B0, 1, B0
        B      .S   loop
```

5. Make the branch conditional based on the value in the loop counter

- What is the syntax for making instruction conditional?

  [condition] Instruction Label

  e.g.

  [B1] B loop

(1) The condition can be one of the following registers: A1, A2, B0, B1, B2.

(2) Any instruction can be conditional.

5. Make the branch conditional based on the value in the loop counter

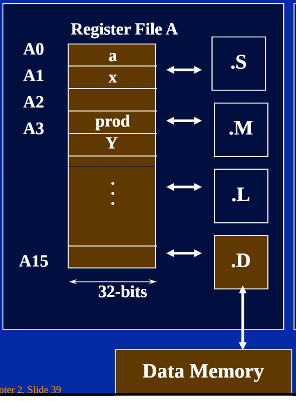
- The condition can be inverted by adding the exclamation symbol "!" as follows:

  [!condition] Instruction Label

  e.g.

  ```
  [!B0]   B   loop   ;branch if B0 = 0
  [B0]    B   loop   ;branch if B0 != 0
  ```

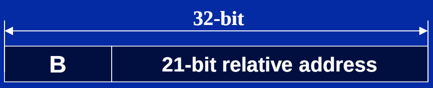
5. Make the branch conditional



```
        MVKL  .S2  pt1, A5
        MVKH  .S2  pt1, A5
        MVKL  .S2  pt2, A6
        MVKH  .S2  pt2, A6
        MVKL  .S2  count, B0
loop    LDH   .D   *A5, A0
        LDH   .D   *A6, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .S   B0, 1, B0
[B0]    B     .S   loop
```

More on the Branch Instruction (1)

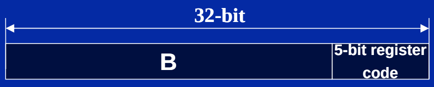
- With this processor all the instructions are encoded in a 32-bit.
- Therefore the label must have a dynamic range of less than 32-bit as the instruction B has to be coded.



- Case 1: B .S1 *label*
  - Relative branch.
  - Label limited to +/- 2^{20} offset.

More on the Branch Instruction (2)

- By specifying a register as an operand instead of a label, it is possible to have an absolute branch.
- This will allow a dynamic range of 2^{32}.



- Case 2: B .S2 *register*
  - Absolute branch.
  - Operates on .S2 ONLY!

Testing the code

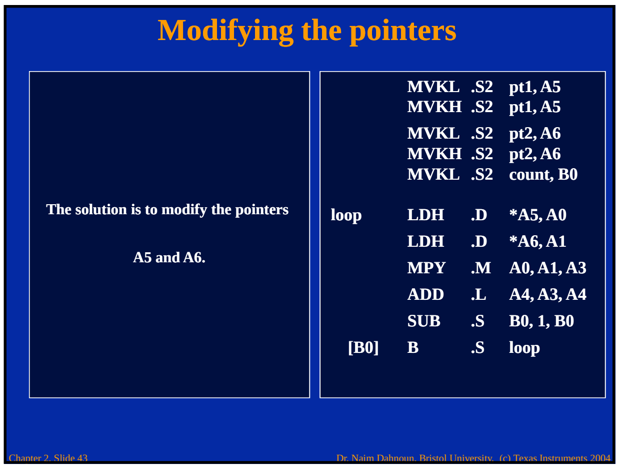
This code performs the following operations:

$a_0*x_0 + a_0*x_0 + a_0*x_0 + \ldots + a_0*x_0$

However, we would like to perform:

$a_0*x_0 + a_1*x_1 + a_2*x_2 + \ldots + a_N*x_N$

```
        MVKL  .S2  pt1, A5
        MVKH  .S2  pt1, A5
        MVKL  .S2  pt2, A6
        MVKH  .S2  pt2, A6
        MVKL  .S2  count, B0
loop    LDH   .D   *A5, A0
        LDH   .D   *A6, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .S   B0, 1, B0
[B0]    B     .S   loop
```

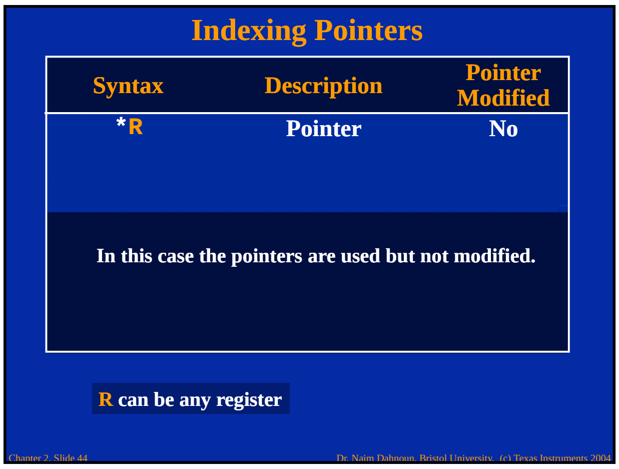


Modifying the pointers

The solution is to modify the pointers

A5 and A6.

```
        MVKL  .S2  pt1, A5
        MVKH  .S2  pt1, A5

        MVKL  .S2  pt2, A6
        MVKH  .S2  pt2, A6
        MVKL  .S2  count, B0

loop    LDH   .D   *A5, A0
        LDH   .D   *A6, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .S   B0, 1, B0
  [B0]  B     .S   loop
```

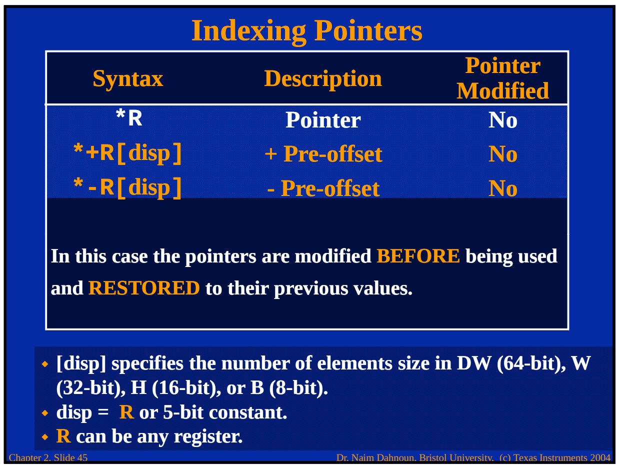


Indexing Pointers

Syntax	Description	Pointer Modified
*R	Pointer	No

In this case the pointers are used but not modified.

R can be any register

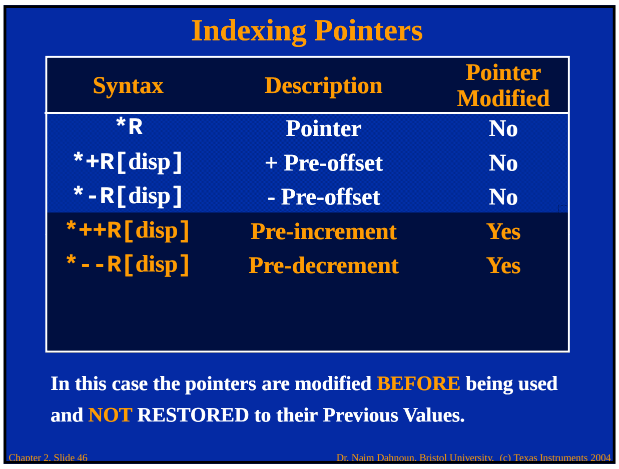


Indexing Pointers

Syntax	Description	Pointer Modified
*R	Pointer	No
*+R[disp]	+ Pre-offset	No
*-R[disp]	- Pre-offset	No

In this case the pointers are modified BEFORE being used and RESTORED to their previous values.

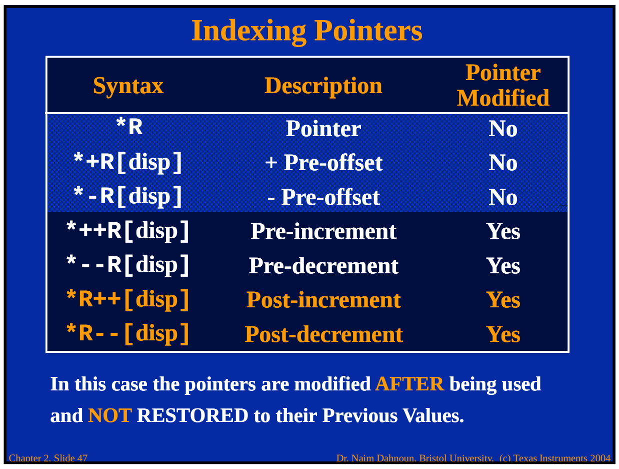
- [disp] specifies the number of elements size in DW (64-bit), W (32-bit), H (16-bit), or B (8-bit).
- disp = R or 5-bit constant.
- R can be any register.



Indexing Pointers

Syntax	Description	Pointer Modified
*R	Pointer	No
*+R[disp]	+ Pre-offset	No
*-R[disp]	- Pre-offset	No
*++R[disp]	Pre-increment	Yes
*--R[disp]	Pre-decrement	Yes

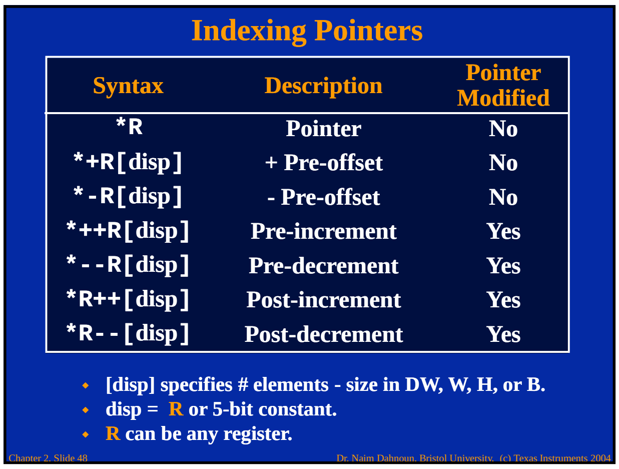
In this case the pointers are modified BEFORE being used and NOT RESTORED to their Previous Values.



Indexing Pointers

Syntax	Description	Pointer Modified
*R	Pointer	No
*+R[disp]	+ Pre-offset	No
*-R[disp]	- Pre-offset	No
*++R[disp]	Pre-increment	Yes
*--R[disp]	Pre-decrement	Yes
*R++[disp]	Post-increment	Yes
*R--[disp]	Post-decrement	Yes

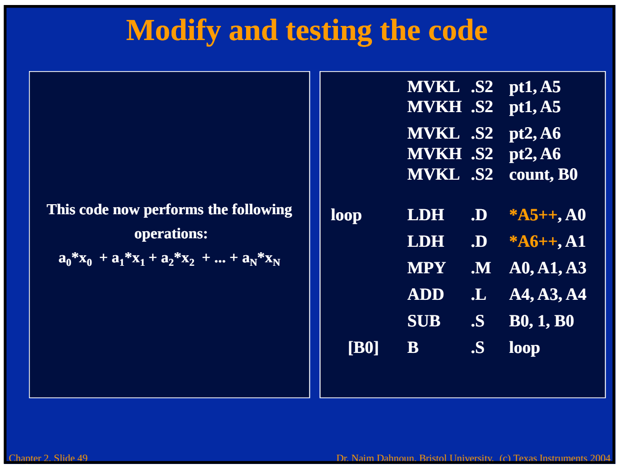
In this case the pointers are modified AFTER being used and NOT RESTORED to their Previous Values.



Indexing Pointers

Syntax	Description	Pointer Modified
*R	Pointer	No
*+R[disp]	+ Pre-offset	No
*-R[disp]	- Pre-offset	No
*++R[disp]	Pre-increment	Yes
*--R[disp]	Pre-decrement	Yes
*R++[disp]	Post-increment	Yes
*R--[disp]	Post-decrement	Yes

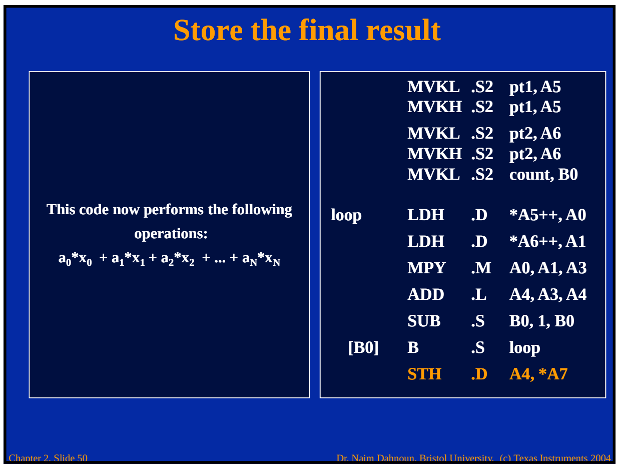
- [disp] specifies # elements - size in DW, W, H, or B.
- disp = R or 5-bit constant.
- R can be any register.



Modify and testing the code

This code now performs the following operations:
$a_0*x_0 + a_1*x_1 + a_2*x_2 + ... + a_N*x_N$

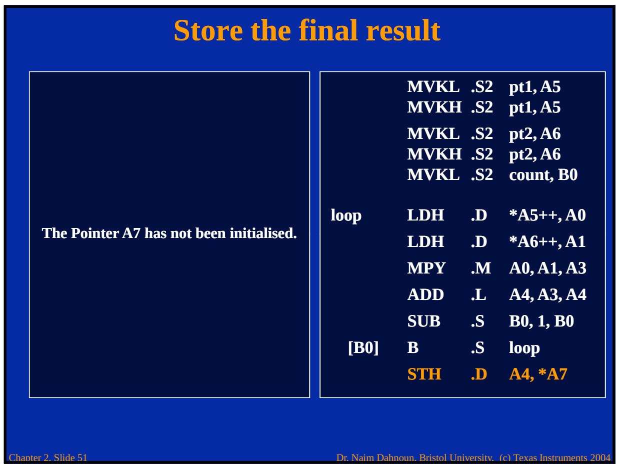
```
        MVKL  .S2  pt1, A5
        MVKH  .S2  pt1, A5

        MVKL  .S2  pt2, A6
        MVKH  .S2  pt2, A6
        MVKL  .S2  count, B0

loop    LDH   .D   *A5++, A0
        LDH   .D   *A6++, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .S   B0, 1, B0
  [B0]  B     .S   loop
```



Store the final result

This code now performs the following operations:
$a_0*x_0 + a_1*x_1 + a_2*x_2 + ... + a_N*x_N$

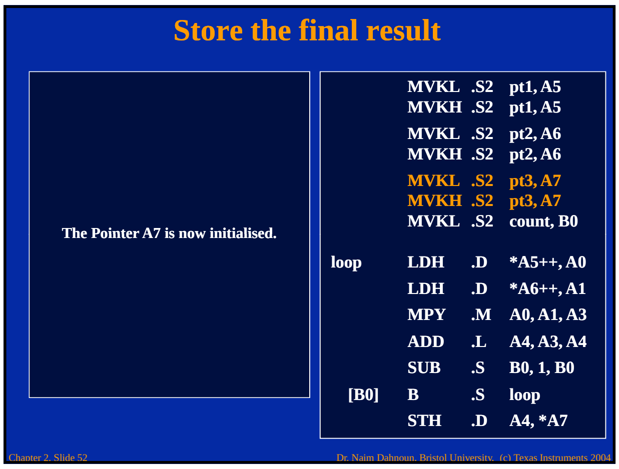
```
        MVKL  .S2  pt1, A5
        MVKH  .S2  pt1, A5

        MVKL  .S2  pt2, A6
        MVKH  .S2  pt2, A6
        MVKL  .S2  count, B0

loop    LDH   .D   *A5++, A0
        LDH   .D   *A6++, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .S   B0, 1, B0
  [B0]  B     .S   loop
        STH   .D   A4, *A7
```



Store the final result

The Pointer A7 has not been initialised.

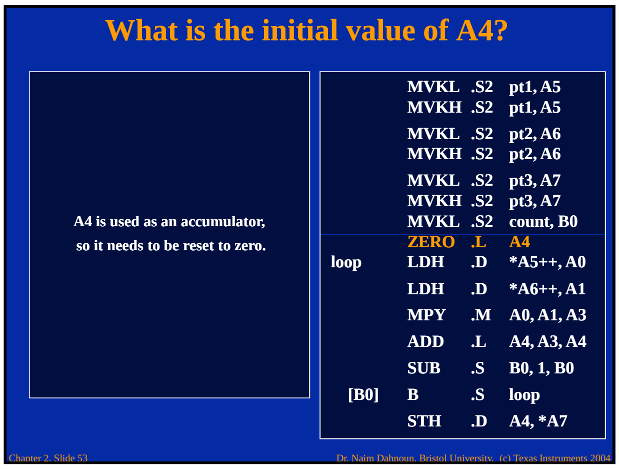
```
        MVKL  .S2  pt1, A5
        MVKH  .S2  pt1, A5

        MVKL  .S2  pt2, A6
        MVKH  .S2  pt2, A6
        MVKL  .S2  count, B0

loop    LDH   .D   *A5++, A0
        LDH   .D   *A6++, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .S   B0, 1, B0
  [B0]  B     .S   loop
        STH   .D   A4, *A7
```



Store the final result

The Pointer A7 is now initialised.

```
        MVKL  .S2  pt1, A5
        MVKH  .S2  pt1, A5

        MVKL  .S2  pt2, A6
        MVKH  .S2  pt2, A6

        MVKL  .S2  pt3, A7
        MVKH  .S2  pt3, A7
        MVKL  .S2  count, B0

loop    LDH   .D   *A5++, A0
        LDH   .D   *A6++, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .S   B0, 1, B0
  [B0]  B     .S   loop
        STH   .D   A4, *A7
```

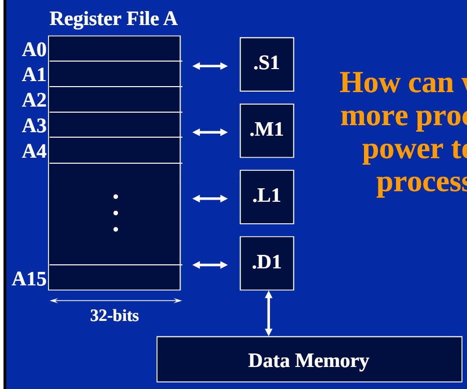


What is the initial value of A4?

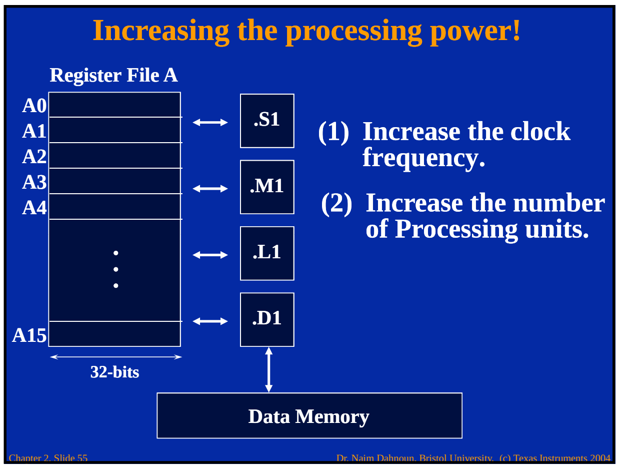
A4 is used as an accumulator,
so it needs to be reset to zero.

```
        MVKL  .S2  pt1, A5
        MVKH  .S2  pt1, A5

        MVKL  .S2  pt2, A6
        MVKH  .S2  pt2, A6

        MVKL  .S2  pt3, A7
        MVKH  .S2  pt3, A7
        MVKL  .S2  count, B0
        ZERO  .L   A4
loop    LDH   .D   *A5++, A0
        LDH   .D   *A6++, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .S   B0, 1, B0
  [B0]  B     .S   loop
        STH   .D   A4, *A7
```

Increasing the processing power!



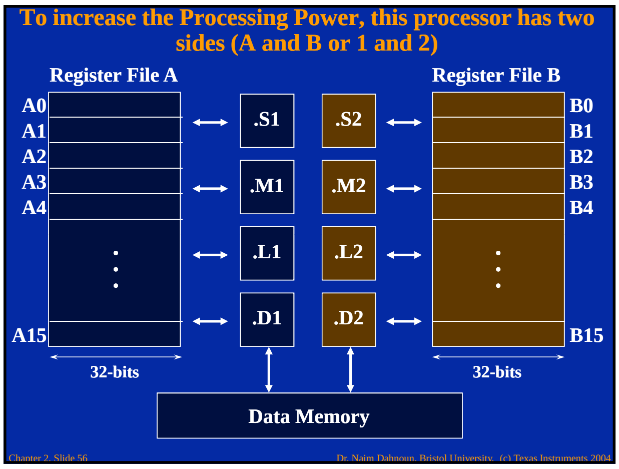
How can we add more processing power to this processor?



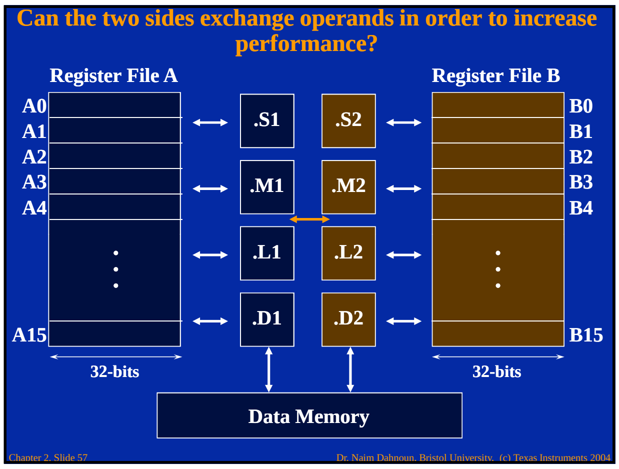
Increasing the processing power!

(1) Increase the clock frequency.

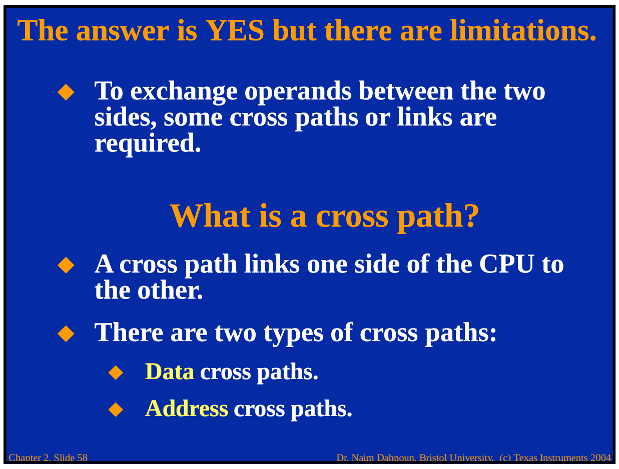
(2) Increase the number of Processing units.



To increase the Processing Power, this processor has two sides (A and B or 1 and 2)



Can the two sides exchange operands in order to increase performance?

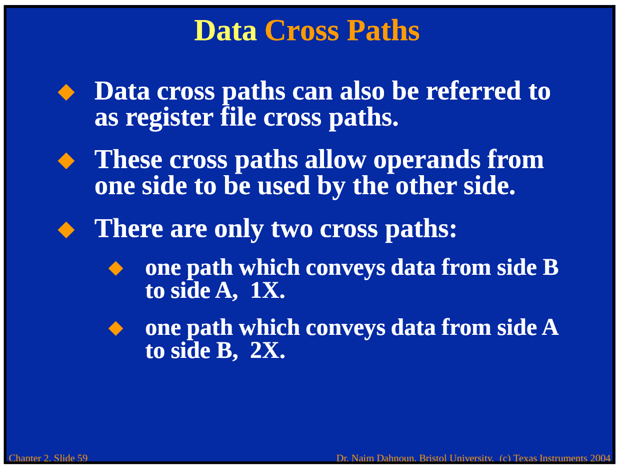


The answer is YES but there are limitations.

- To exchange operands between the two sides, some cross paths or links are required.

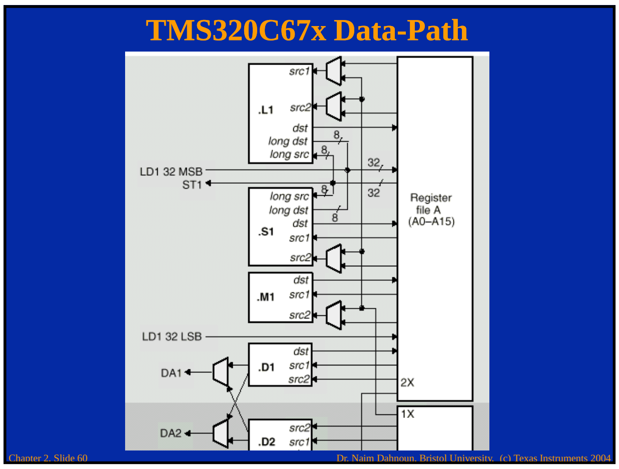
What is a cross path?

- A cross path links one side of the CPU to the other.
- There are two types of cross paths:
  - Data cross paths.
  - Address cross paths.

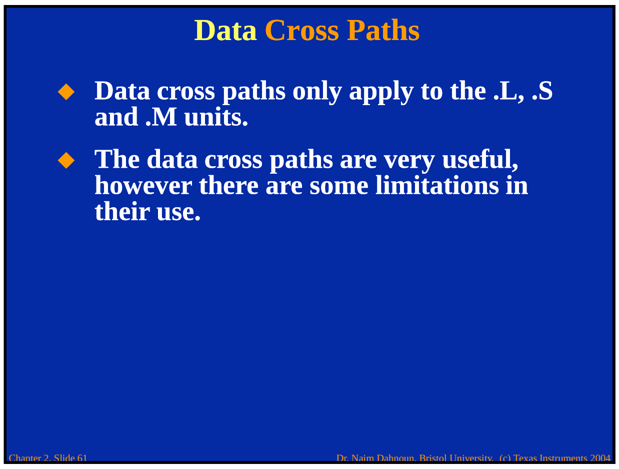


Data Cross Paths

- Data cross paths can also be referred to as register file cross paths.
- These cross paths allow operands from one side to be used by the other side.
- There are only two cross paths:
  - one path which conveys data from side B to side A, 1X.
  - one path which conveys data from side A to side B, 2X.

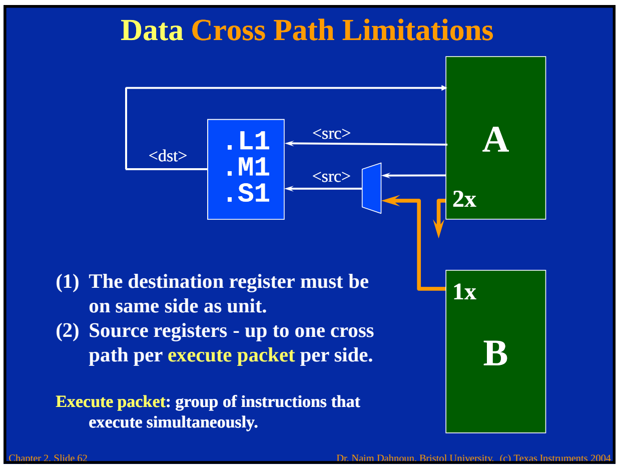


TMS320C67x Data-Path



Data Cross Paths

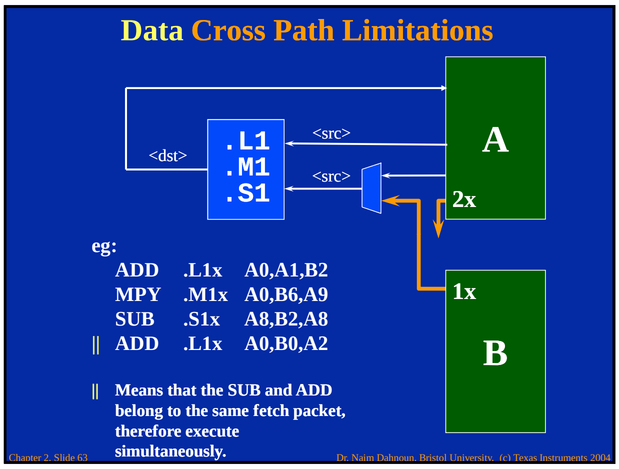
- Data cross paths only apply to the .L, .S and .M units.
- The data cross paths are very useful, however there are some limitations in their use.



Data Cross Path Limitations

(1) The destination register must be on same side as unit.
(2) Source registers - up to one cross path per execute packet per side.

Execute packet: group of instructions that execute simultaneously.

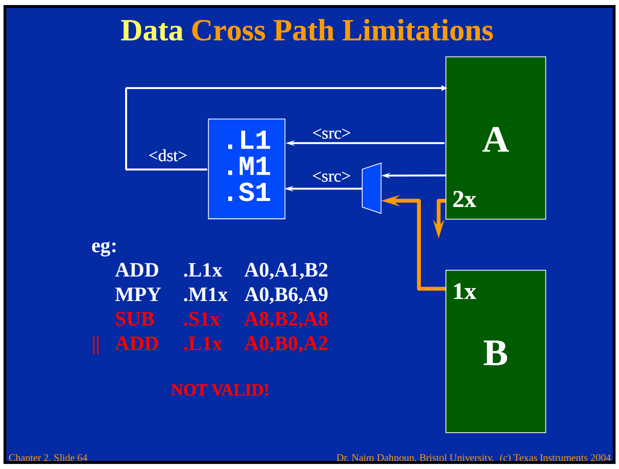


Data Cross Path Limitations

eg:

```
   ADD   .L1x   A0,A1,B2
   MPY   .M1x   A0,B6,A9
   SUB   .S1x   A8,B2,A8
|| ADD   .L1x   A0,B0,A2
```

|| Means that the SUB and ADD belong to the same fetch packet, therefore execute simultaneously.

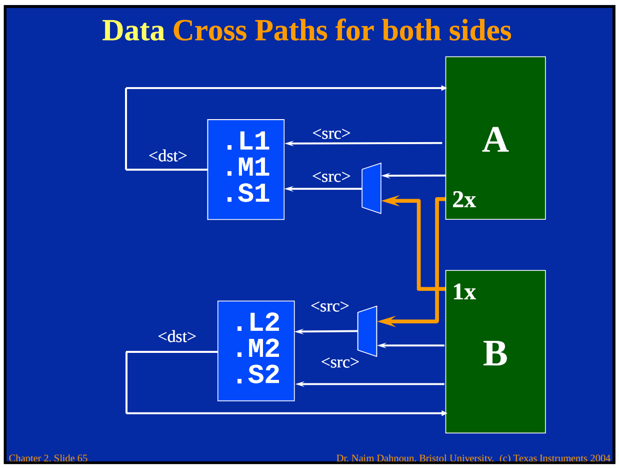


Data Cross Path Limitations

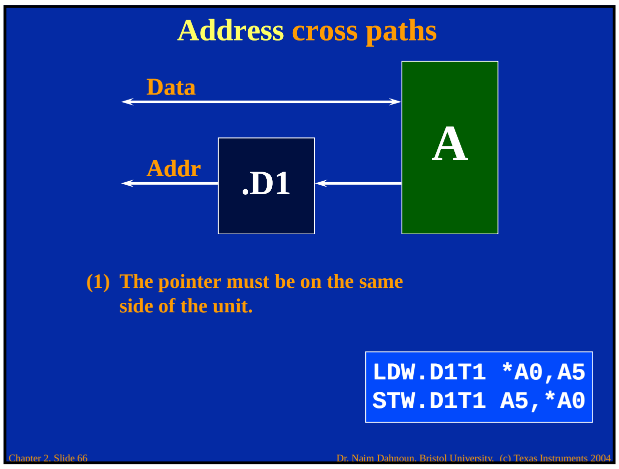
eg:

```
   ADD   .L1x   A0,A1,B2
   MPY   .M1x   A0,B6,A9
   SUB   .S1x   A8,B2,A8
|| ADD   .L1x   A0,B0,A2
```

NOT VALID!



Data Cross Paths for both sides

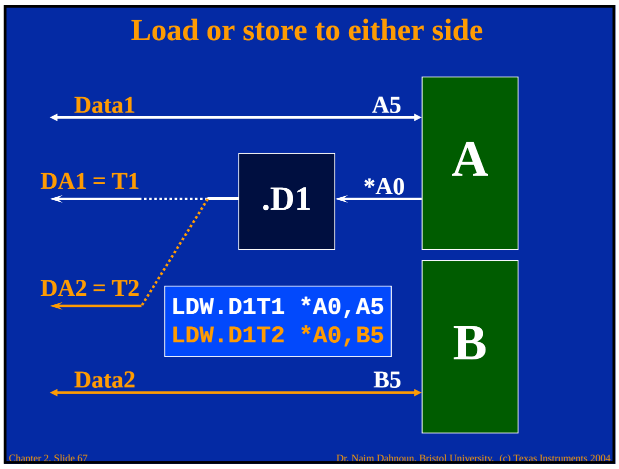


Address cross paths

(1) The pointer must be on the same side of the unit.

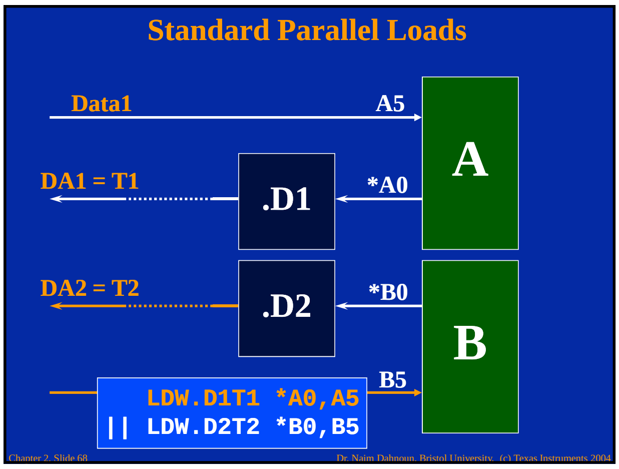
```
LDW.D1T1 *A0,A5
STW.D1T1 A5,*A0
```

Load or store to either side



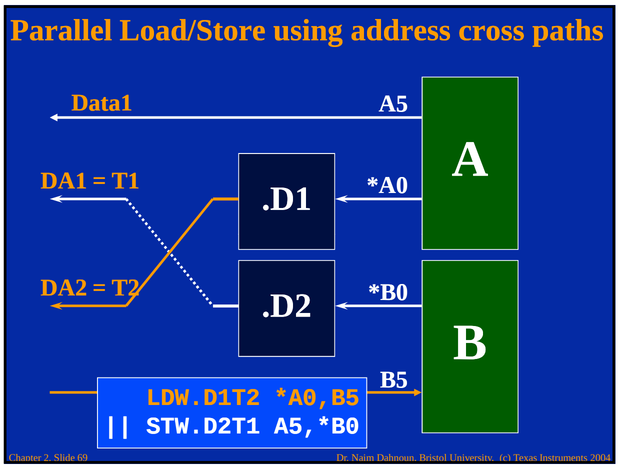
```
LDW.D1T1 *A0,A5
LDW.D1T2 *A0,B5
```

Standard Parallel Loads



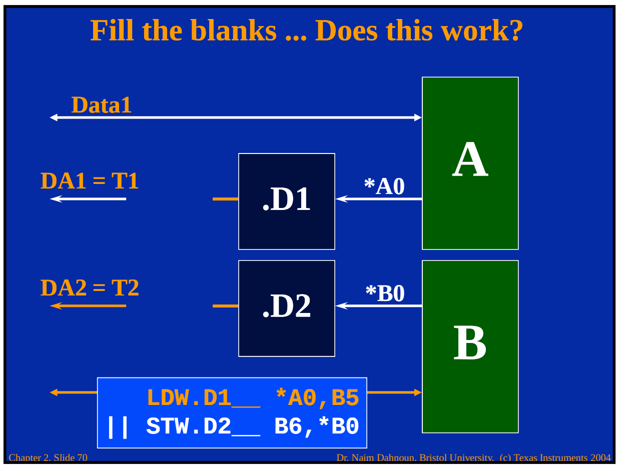
```
   LDW.D1T1 *A0,A5
|| LDW.D2T2 *B0,B5
```

Parallel Load/Store using address cross paths



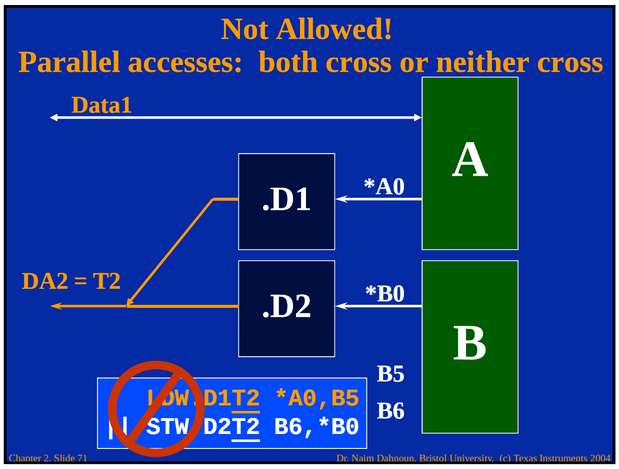
```
   LDW.D1T2 *A0,B5
|| STW.D2T1 A5,*B0
```

Fill the blanks ... Does this work?



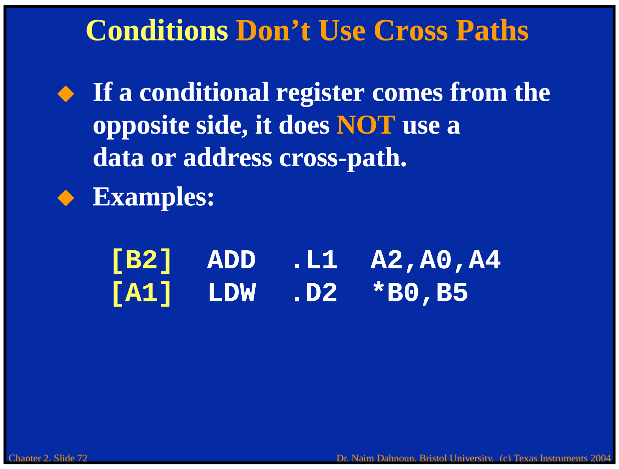
```
   LDW.D1__ *A0,B5
|| STW.D2__ B6,*B0
```

Not Allowed! Parallel accesses: both cross or neither cross



```
   LDW.D1T2 *A0,B5
|| STW.D2T2 B6,*B0
```

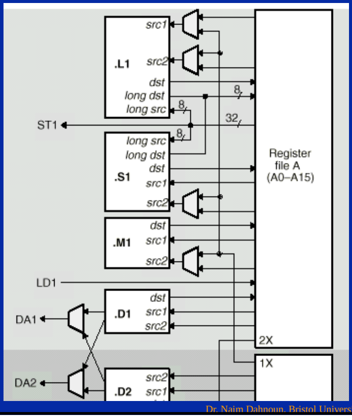
Conditions Don't Use Cross Paths



- If a conditional register comes from the opposite side, it does NOT use a data or address cross-path.
- Examples:

```
[B2]  ADD  .L1  A2,A0,A4
[A1]  LDW  .D2  *B0,B5
```

‘C62x Data-Path Summary

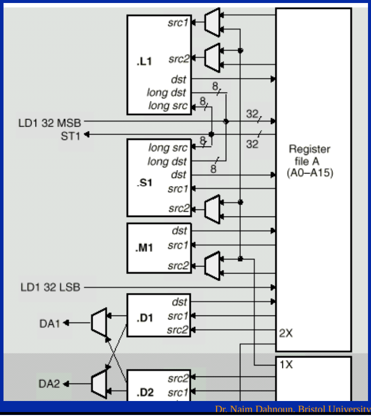


CPU Ref Guide

Full CPU Datapath (Pg 2-2)

‘C67x Data-Path Summary

‘C67x



Cross Paths - Summary

- ✓ **Data**
  - **Destination register on same side as unit.**
  - **Source registers - up to one cross path per execute packet per side.**
  - **Use “x” to indicate cross-path.**
- ✓ **Address**
  - **Pointer must be on same side as unit.**
  - **Data can be transferred to/from either side.**
  - **Parallel accesses: both cross or neither cross.**
- ✓ **Conditionals Don’t Use Cross Paths.**

Code Review (using side A only)

$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
         MVK   .S1   40, A2       ; A2 = 40, loop count
loop:    LDH   .D1   *A5++, A0    ; A0 = a(n)
         LDH   .D1   *A6++, A1    ; A1 = x(n)
         MPY   .M1   A0, A1, A3   ; A3 = a(n) * x(n)
         ADD   .L1   A3, A4, A4   ; Y = Y + A3
         SUB   .L1   A2, 1, A2    ; decrement loop count
  [A2]   B     .S1   loop         ; if A2 ≠ 0, branch
         STH   .D1   A4, *A7      ; *A7 = Y
```

Note: Assume that A4 was previously cleared and the pointers are initialised.

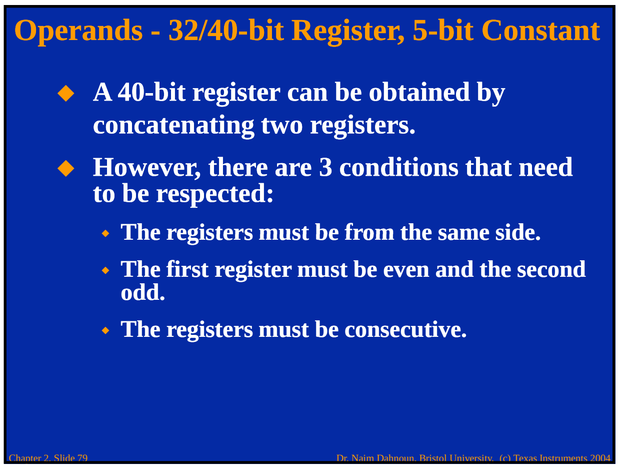
Let us have a look at the final details concerning the functional units.

Consider first the case of the .L and .S units.

Operands - 32/40-bit Register, 5-bit Constant

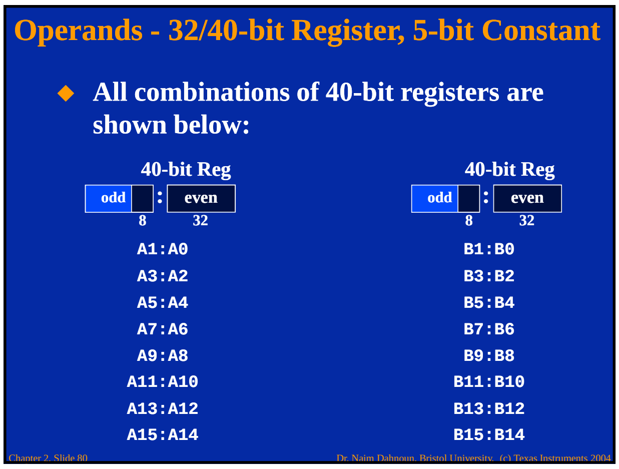
- **Operands can be:**
  - **5-bit constants (or 16-bit for MVKL and MVKH).**
  - **32-bit registers.**
  - **40-bit Registers.**
- **However, we have seen that registers are only 32-bit.**

So where do the 40-bit registers come from?



Operands - 32/40-bit Register, 5-bit Constant

- A 40-bit register can be obtained by concatenating two registers.
- However, there are 3 conditions that need to be respected:
  - The registers must be from the same side.
  - The first register must be even and the second odd.
  - The registers must be consecutive.

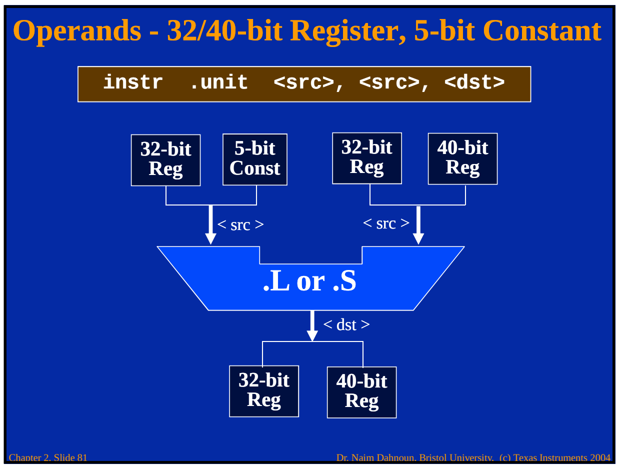


Operands - 32/40-bit Register, 5-bit Constant

- All combinations of 40-bit registers are shown below:

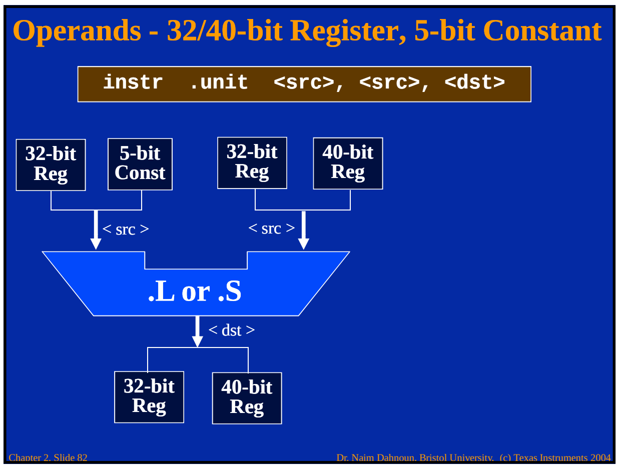
40-bit Reg	40-bit Reg
A1:A0	B1:B0
A3:A2	B3:B2
A5:A4	B5:B4
A7:A6	B7:B6
A9:A8	B9:B8
A11:A10	B11:B10
A13:A12	B13:B12
A15:A14	B15:B14

odd
:
even
8
32



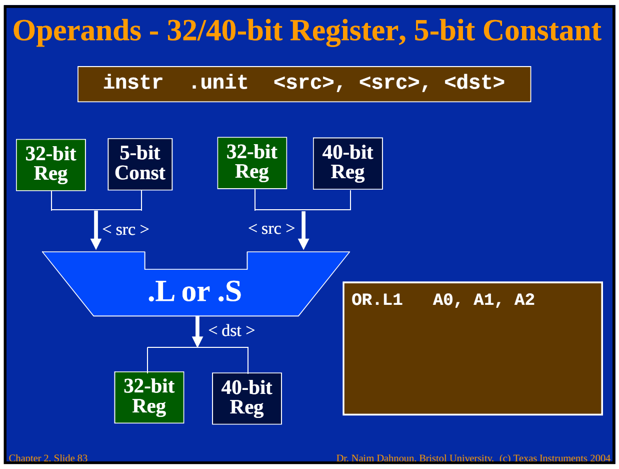
Operands - 32/40-bit Register, 5-bit Constant

```
instr  .unit  <src>, <src>, <dst>
```



Operands - 32/40-bit Register, 5-bit Constant

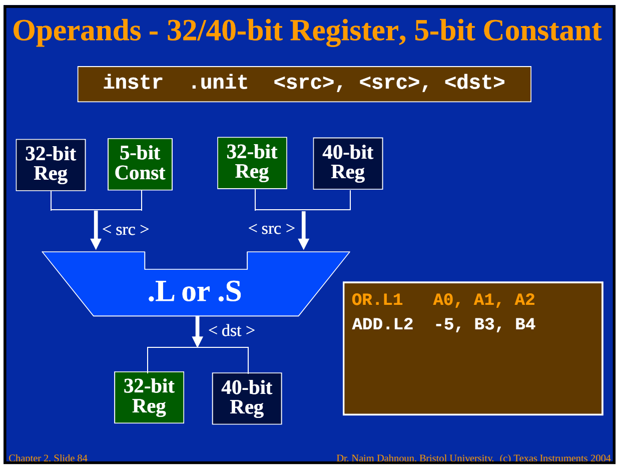
```
instr  .unit  <src>, <src>, <dst>
```



Operands - 32/40-bit Register, 5-bit Constant

```
instr  .unit  <src>, <src>, <dst>
```

```
OR.L1   A0, A1, A2
```



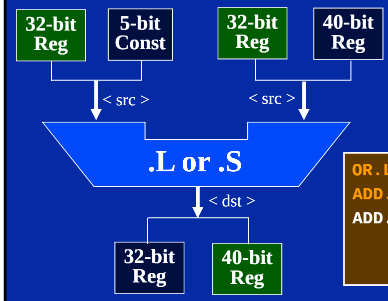
Operands - 32/40-bit Register, 5-bit Constant

```
instr  .unit  <src>, <src>, <dst>
```

```
OR.L1   A0, A1, A2
ADD.L2  -5, B3, B4
```

Operands - 32/40-bit Register, 5-bit Constant

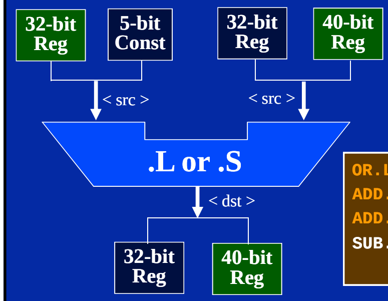
```
instr  .unit  <src>, <src>, <dst>
```



```
OR.L1   A0, A1, A2
ADD.L2  -5, B3, B4
ADD.L1  A2, A3, A5:A4
```

Operands - 32/40-bit Register, 5-bit Constant

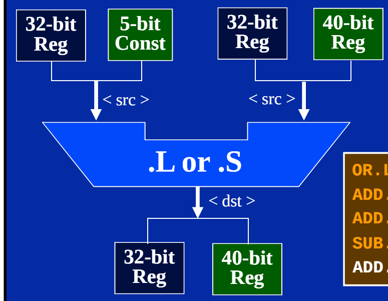
```
instr  .unit  <src>, <src>, <dst>
```



```
OR.L1   A0, A1, A2
ADD.L2  -5, B3, B4
ADD.L1  A2, A3, A5:A4
SUB.L1  A2, A5:A4, A5:A4
```

Operands - 32/40-bit Register, 5-bit Constant

```
instr  .unit  <src>, <src>, <dst>
```



```
OR.L1   A0, A1, A2
ADD.L2  -5, B3, B4
ADD.L1  A2, A3, A5:A4
SUB.L1  A2, A5:A4, A5:A4
ADD.L2  3, B9:B8, B9:B8
```

Register to register data transfer

- To move the content of a register (A or B) to another register (B or A) use the move "MV" Instruction, e.g.:

```
MV    A0, B0
MV    B6, B7
```

- To move the content of a control register to another register (A or B) or vice-versa use the MVC instruction, e.g.:

```
MVC   IFR, A0
MVC   A0, IRP
```

TMS320C6000 Instruction Set

'C62x Instruction Set (by category)

Arithmetic
ABS
ADD
ADDA
ADDK
ADD2
MPY
MPYH
NEG
SMPY
SMPYH
SADD
SAT
SSUB
SUB
SUBA
SUBC
SUB2
ZERO

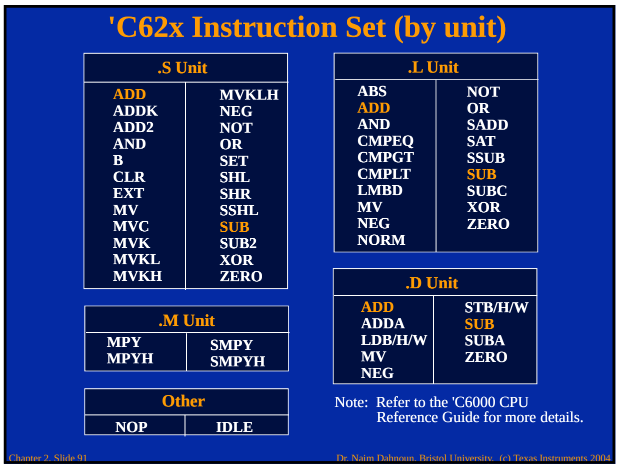
Logical
AND
CMPEQ
CMPGT
CMPLT
NOT
OR
SHL
SHR
SSHL
XOR

Bit Mgmt
CLR
EXT
LMBD
NORM
SET

Data Mgmt
LDB/H/W
MV
MVC
MVK
MVKL
MVKH
MVKLH
STB/H/W

Program Ctrl
B
IDLE
NOP

Note: Refer to the 'C6000 CPU Reference Guide for more details.



'C62x Instruction Set (by unit)

.S Unit

ADD	MVKLH
ADDK	NEG
ADD2	NOT
AND	OR
B	SET
CLR	SHL
EXT	SHR
MV	SSHL
MVC	SUB
MVK	SUB2
MVKL	XOR
MVKH	ZERO

.L Unit

ABS	NOT
ADD	OR
AND	SADD
CMPEQ	SAT
CMPGT	SSUB
CMPLT	SUB
LMBD	SUBC
MV	XOR
NEG	ZERO
NORM	

.M Unit

MPY	SMPY
MPYH	SMPYH

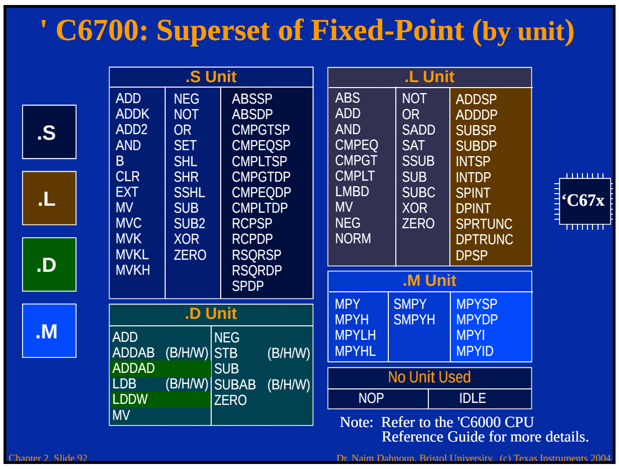
.D Unit

ADD	STB/H/W
ADDA	SUB
LDB/H/W	SUBA
MV	ZERO
NEG	

Other

NOP	IDLE

Note: Refer to the 'C6000 CPU Reference Guide for more details.



' C6700: Superset of Fixed-Point (by unit)

.S Unit

ADD	NEG	ABSSP
ADDK	NOT	ABSDP
ADD2	OR	CMPGTSP
AND	SET	CMPEQSP
B	SHL	CMPLTSP
CLR	SHR	CMPGTDP
EXT	SSHL	CMPEQDP
MV	SUB	CMPLTDP
MVC	SUB2	RCPSP
MVK	XOR	RCPDP
MVKL	ZERO	RSQRSP
MVKH		RSQRDP
		SPDP

.L Unit

ABS	NOT	ADDSP
ADD	OR	ADDDP
AND	SADD	SUBSP
CMPEQ	SAT	SUBDP
CMPGT	SSUB	INTSP
CMPLT	SUB	INTDP
LMBD	SUBC	SPINT
MV	XOR	DPINT
NEG	ZERO	SPRTUNC
NORM		DPTRUNC
		DPSP

.D Unit

ADD	NEG
ADDAB (B/H/W)	STB (B/H/W)
ADDAD	SUB
LDB (B/H/W)	SUBAB (B/H/W)
LDDW	ZERO
MV	

.M Unit

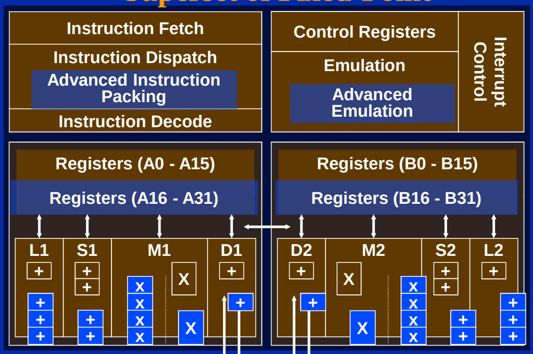
MPY	SMPY	MPYSP
MPYH	SMPYH	MPYDP
MPYLH		MPYI
MPYHL		MPYID

No Unit Used

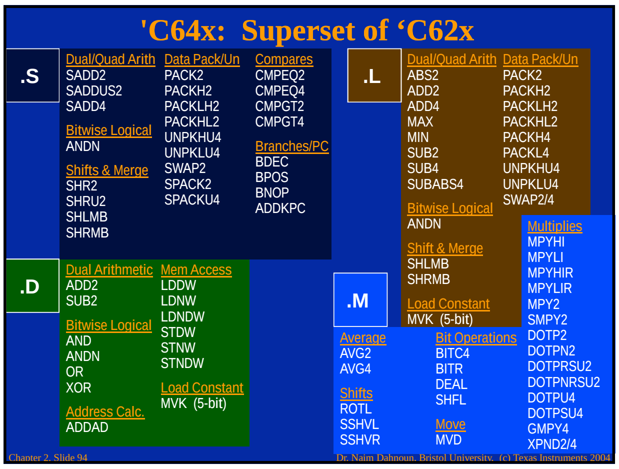
NOP	IDLE

Note: Refer to the 'C6000 CPU Reference Guide for more details.

Superset of Fixed-Point



'C62x: Dual 32-Bit Load/Store
'C64x: Dual 64-Bit Load/Store
'C67x: Dual 64-Bit Load/32-Bit Store



'C64x: Superset of 'C62x

.S

Dual/Quad Arith: SADD2, SADDUS2, SADD4

Bitwise Logical: ANDN

Shifts & Merge: SHR2, SHRU2, SHLMB, SHRMB

Data Pack/Un: PACK2, PACKH2, PACKLH2, PACKHL2, UNPKHU4, UNPKLU4, SWAP2, SPACK2, SPACKU4

Compares: CMPEQ2, CMPEQ4, CMPGT2, CMPGT4

Branches/PC: BDEC, BPOS, BNOP, ADDKPC

.L

Dual/Quad Arith: ABS2, ADD2, ADD4, MAX, MIN, SUB2, SUB4, SUBABS4

Bitwise Logical: ANDN

Shift & Merge: SHLMB, SHRMB

Load Constant: MVK (5-bit)

Data Pack/Un: PACK2, PACKH2, PACKLH2, PACKHL2, PACKH4, PACKL4, UNPKHU4, UNPKLU4, SWAP2/4

.D

Dual Arithmetic: ADD2, SUB2

Bitwise Logical: AND, ANDN, OR, XOR

Address Calc.: ADDAD

Mem Access: LDDW, LDNW, LDNDW, STDW, STNW, STNDW

Load Constant: MVK (5-bit)

.M

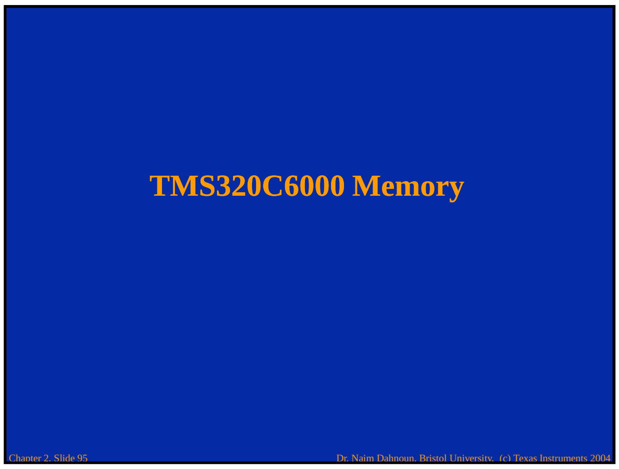
Average: AVG2, AVG4

Shifts: ROTL, SSHVL, SSHVR

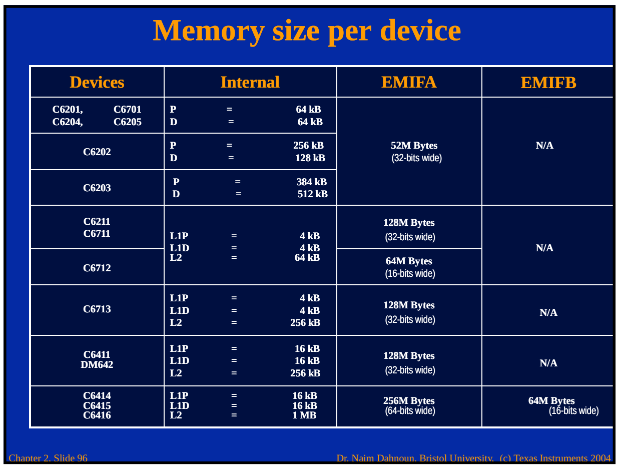
Bit Operations: BITC4, BITR, DEAL, SHFL

Move: MVD

Multiplies: MPYHI, MPYLI, MPYHIR, MPYLIR, MPY2, SMPY2, DOTP2, DOTPN2, DOTPRSU2, DOTPNRSU2, DOTPU4, DOTPSU4, GMPY4, XPND2/4



TMS320C6000 Memory



Memory size per device

Devices	Internal	EMIFA	EMIFB

C6201, C6701 C6204, C6205	P = 64 kB
	D = 64 kB		

C6202	P = 256 kB
	D = 128 kB
		52M Bytes (32-bits wide)	N/A

C6203	P = 384 kB
	D = 512 kB		

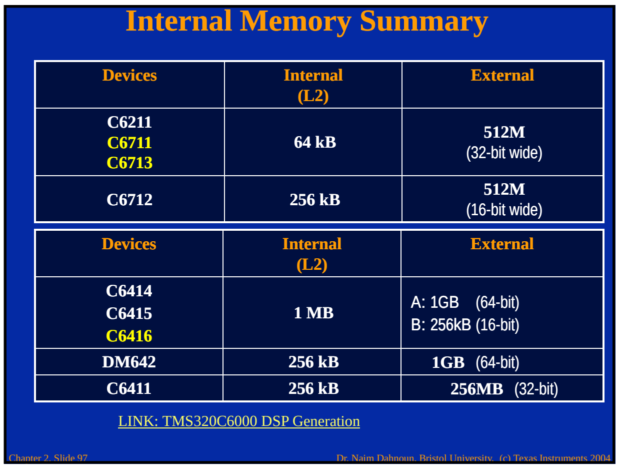
C6211 C6711	L1P = 4 kB
	L1D = 4 kB
	L2 = 64 kB
		128M Bytes (32-bits wide)	N/A

C6712		64M Bytes (16-bits wide)	

C6713	L1P = 4 kB
	L1D = 4 kB
	L2 = 256 kB
		128M Bytes (32-bits wide)	N/A

C6411 DM642	L1P = 16 kB
	L1D = 16 kB
	L2 = 256 kB
		128M Bytes (32-bits wide)	N/A

C6414 C6415 C6416	L1P = 16 kB
	L1D = 16 kB
	L2 = 1 MB
		256M Bytes (64-bits wide)	64M Bytes (16-bits wide)



Internal Memory Summary

Devices	Internal (L2)	External
C6211 C6711 C6713	64 kB	512M (32-bit wide)
C6712	256 kB	512M (16-bit wide)

Devices	Internal (L2)	External
C6414 C6415 C6416	1 MB	A: 1GB (64-bit) B: 256kB (16-bit)
DM642	256 kB	1GB (64-bit)
C6411	256 kB	256MB (32-bit)

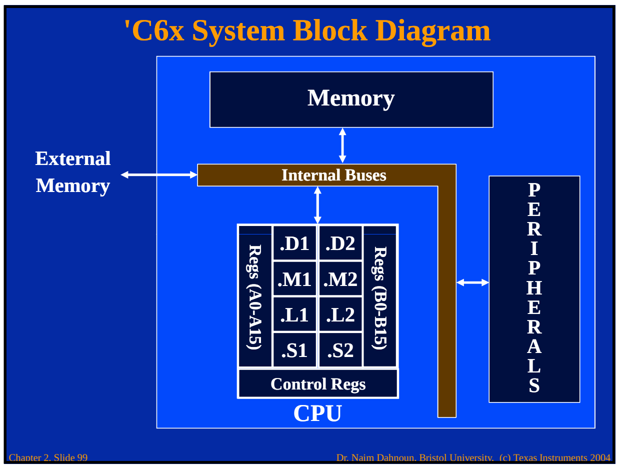
LINK: TMS320C6000 DSP Generation

Chapter 2, Slide 97 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

TMS320C6000 Peripherals

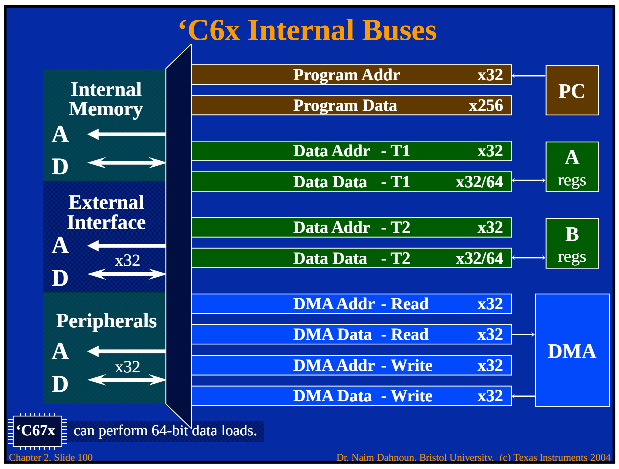
Chapter 2, Slide 98 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

'C6x System Block Diagram



Chapter 2, Slide 99 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

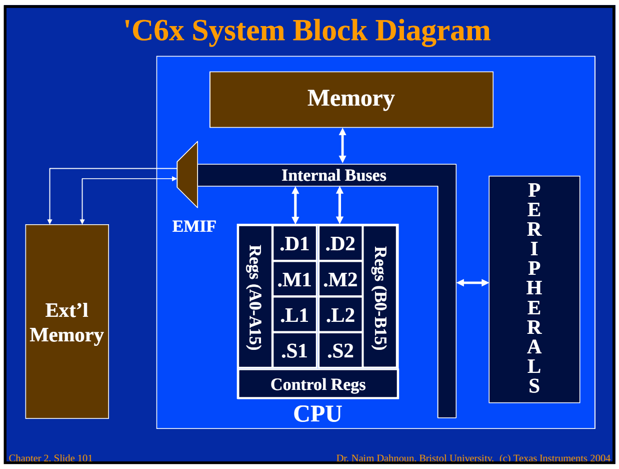
‘C6x Internal Buses



‘C67x can perform 64-bit data loads.

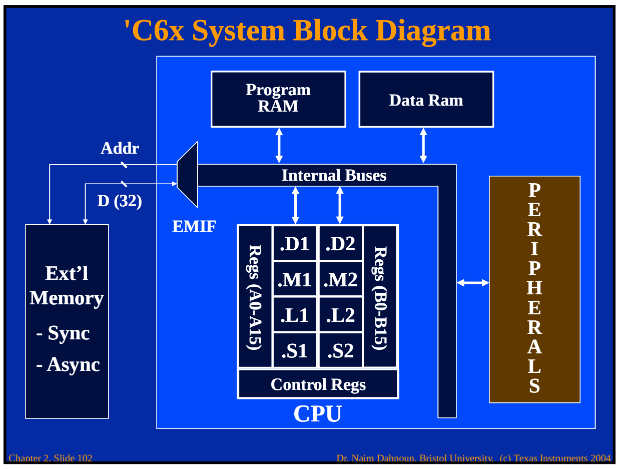
Chapter 2, Slide 100 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

'C6x System Block Diagram

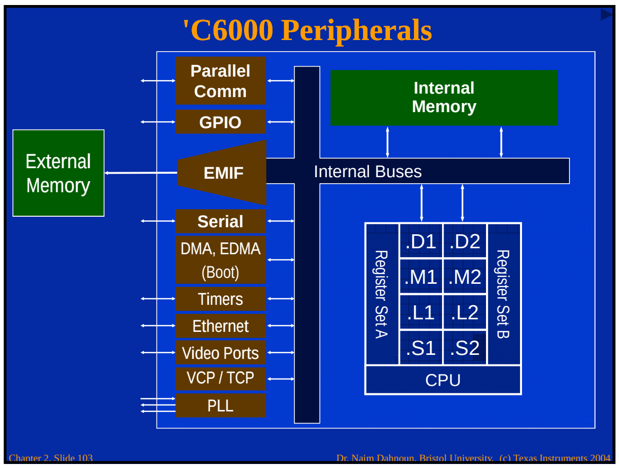


Chapter 2, Slide 101 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

'C6x System Block Diagram

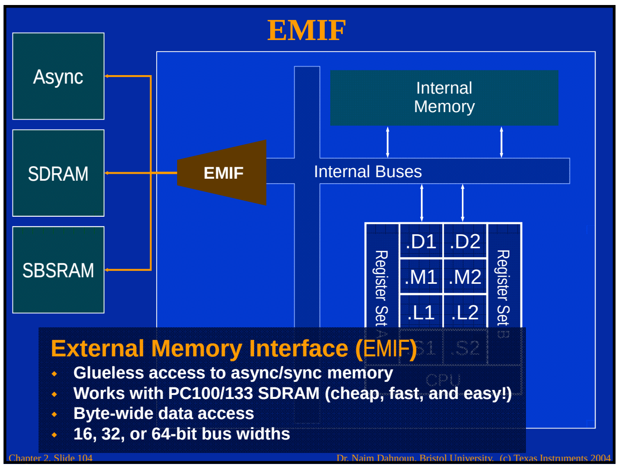


Chapter 2, Slide 102 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004



'C6000 Peripherals

Chapter 2, Slide 103 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

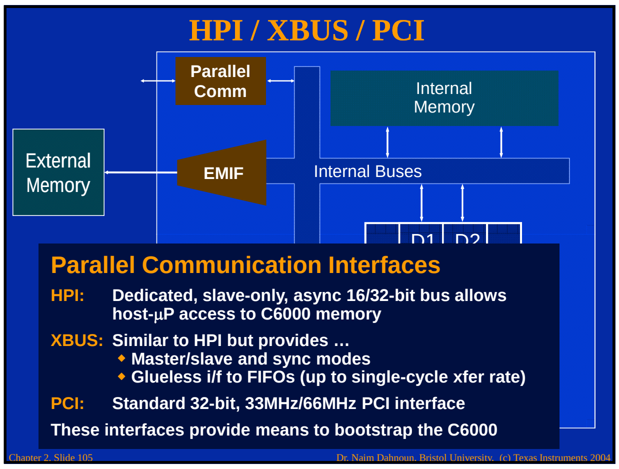


EMIF

External Memory Interface (EMIF)

- Glueless access to async/sync memory
- Works with PC100/133 SDRAM (cheap, fast, and easy!)
- Byte-wide data access
- 16, 32, or 64-bit bus widths

Chapter 2, Slide 104 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004



HPI / XBUS / PCI

Parallel Communication Interfaces

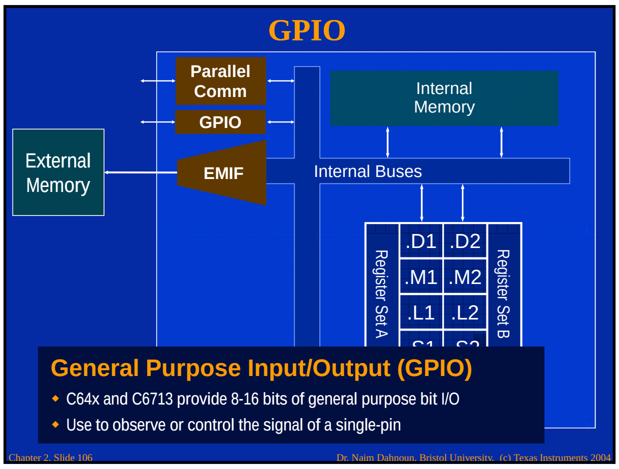
HPI: Dedicated, slave-only, async 16/32-bit bus allows host-μP access to C6000 memory

XBUS: Similar to HPI but provides …
- Master/slave and sync modes
- Glueless i/f to FIFOs (up to single-cycle xfer rate)

PCI: Standard 32-bit, 33MHz/66MHz PCI interface

These interfaces provide means to bootstrap the C6000

Chapter 2, Slide 105 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

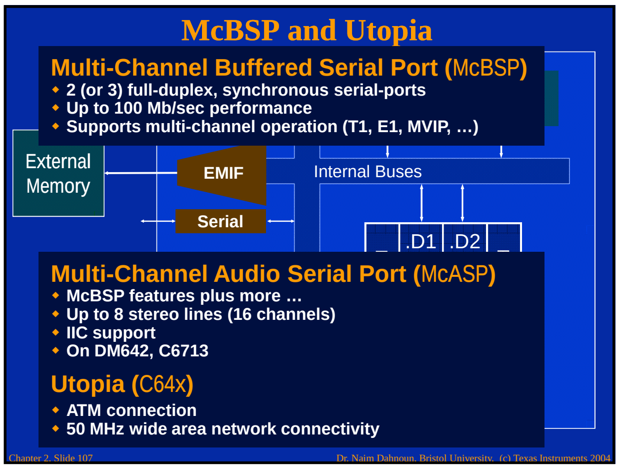


GPIO

General Purpose Input/Output (GPIO)

- C64x and C6713 provide 8-16 bits of general purpose bit I/O
- Use to observe or control the signal of a single-pin

Chapter 2, Slide 106 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004



McBSP and Utopia

Multi-Channel Buffered Serial Port (McBSP)

- 2 (or 3) full-duplex, synchronous serial-ports
- Up to 100 Mb/sec performance
- Supports multi-channel operation (T1, E1, MVIP, …)

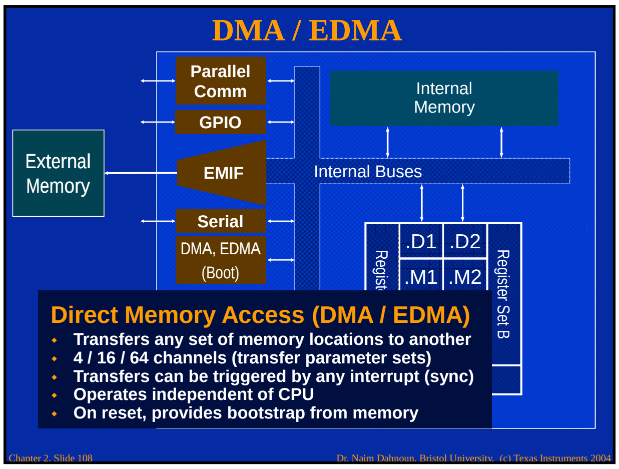
Multi-Channel Audio Serial Port (McASP)

- McBSP features plus more …
- Up to 8 stereo lines (16 channels)
- IIC support
- On DM642, C6713

Utopia (C64x)

- ATM connection
- 50 MHz wide area network connectivity

Chapter 2, Slide 107 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

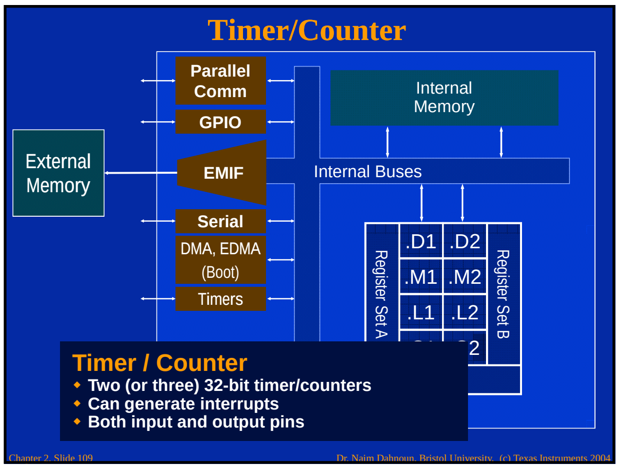


DMA / EDMA

Direct Memory Access (DMA / EDMA)

- Transfers any set of memory locations to another
- 4 / 16 / 64 channels (transfer parameter sets)
- Transfers can be triggered by any interrupt (sync)
- Operates independent of CPU
- On reset, provides bootstrap from memory

Chapter 2, Slide 108 Dr. Naim Dahnoun, Bristol University, (c) Texas Instruments 2004

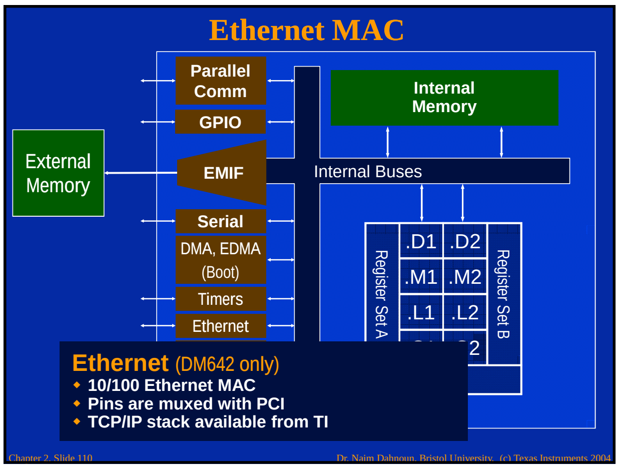


Parallel Comm
GPIO
External Memory
EMIF
Internal Memory
Internal Buses
Serial
DMA, EDMA (Boot)
Timers
Register Set A
Register Set B
.D1 .D2
.M1 .M2
.L1 .L2

Timer/Counter

Timer / Counter

- **Two (or three) 32-bit timer/counters**
- **Can generate interrupts**
- **Both input and output pins**

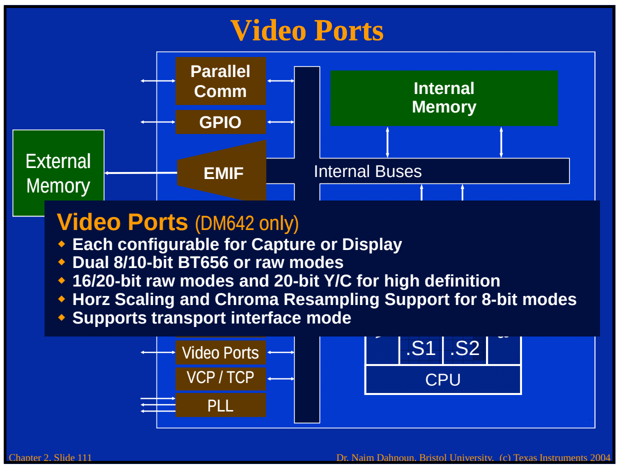


Parallel Comm
GPIO
External Memory
EMIF
Internal Memory
Internal Buses
Serial
DMA, EDMA (Boot)
Timers
Ethernet
Register Set A
Register Set B
.D1 .D2
.M1 .M2
.L1 .L2

Ethernet MAC

Ethernet (DM642 only)

- **10/100 Ethernet MAC**
- **Pins are muxed with PCI**
- **TCP/IP stack available from TI**

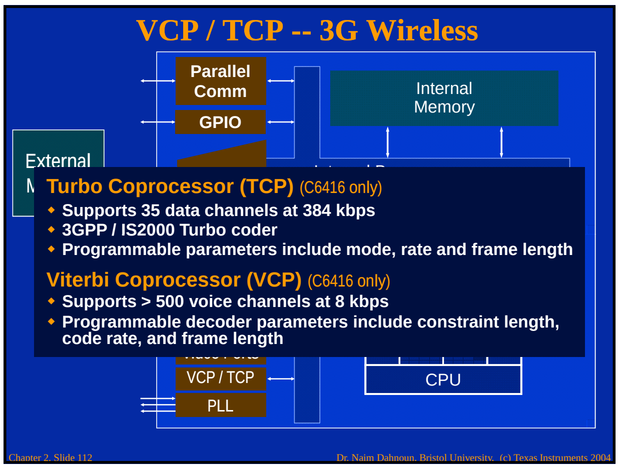


Parallel Comm
GPIO
External Memory
EMIF
Internal Memory
Internal Buses
Video Ports
VCP / TCP
PLL
.S1 .S2
CPU

Video Ports

Video Ports (DM642 only)

- **Each configurable for Capture or Display**
- **Dual 8/10-bit BT656 or raw modes**
- **16/20-bit raw modes and 20-bit Y/C for high definition**
- **Horz Scaling and Chroma Resampling Support for 8-bit modes**
- **Supports transport interface mode**



Parallel Comm
GPIO
External
Internal Memory
VCP / TCP
PLL
CPU

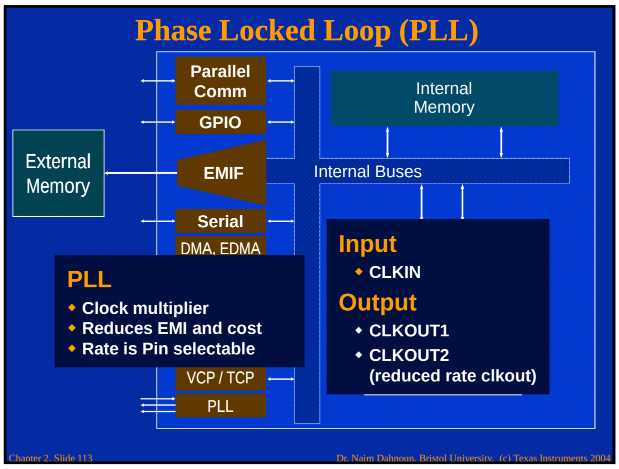
VCP / TCP -- 3G Wireless

Turbo Coprocessor (TCP) (C6416 only)

- **Supports 35 data channels at 384 kbps**
- **3GPP / IS2000 Turbo coder**
- **Programmable parameters include mode, rate and frame length**

Viterbi Coprocessor (VCP) (C6416 only)

- **Supports > 500 voice channels at 8 kbps**
- **Programmable decoder parameters include constraint length, code rate, and frame length**



Parallel Comm
GPIO
External Memory
EMIF
Internal Memory
Internal Buses
Serial
DMA, EDMA
VCP / TCP
PLL

Phase Locked Loop (PLL)

PLL

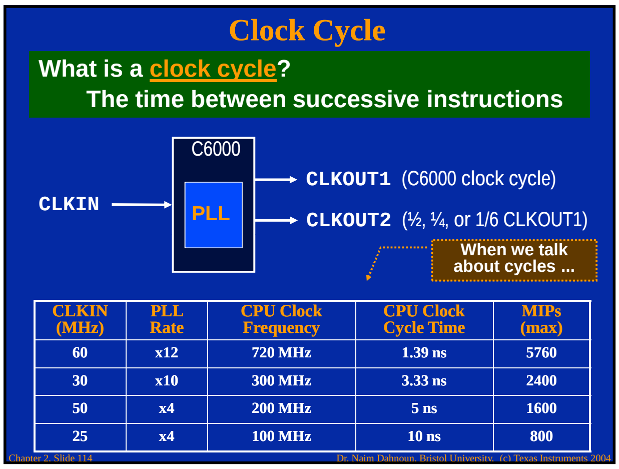
- **Clock multiplier**
- **Reduces EMI and cost**
- **Rate is Pin selectable**

Input

- **CLKIN**

Output

- **CLKOUT1**
- **CLKOUT2 (reduced rate clkout)**

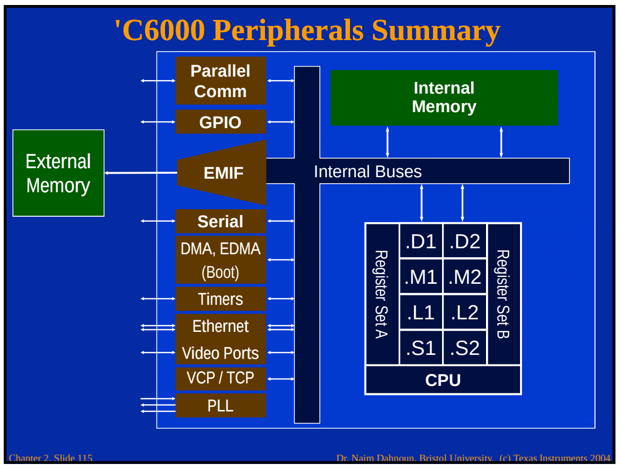


C6000
CLKIN
PLL
CLKOUT1 (C6000 clock cycle)
CLKOUT2 (½, ¼, or 1/6 CLKOUT1)
When we talk about cycles ...

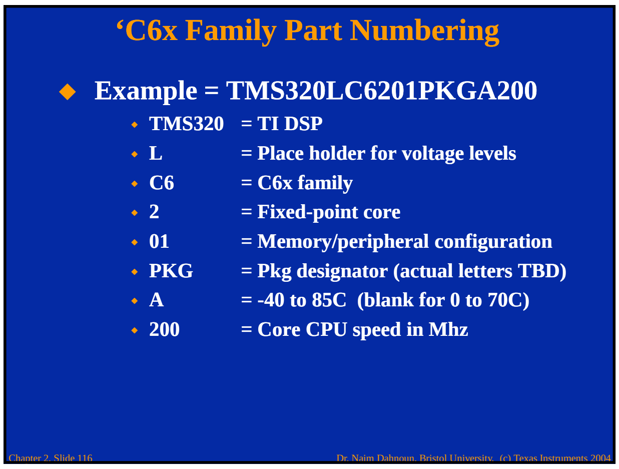
Clock Cycle

What is a clock cycle?
The time between successive instructions

CLKIN (MHz)	PLL Rate	CPU Clock Frequency	CPU Clock Cycle Time	MIPs (max)
60	x12	720 MHz	1.39 ns	5760
30	x10	300 MHz	3.33 ns	2400
50	x4	200 MHz	5 ns	1600
25	x4	100 MHz	10 ns	800



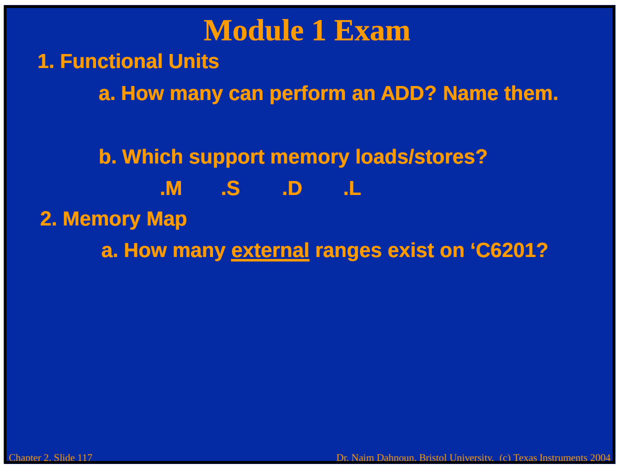
'C6000 Peripherals Summary



'C6x Family Part Numbering

- Example = TMS320LC6201PKGA200

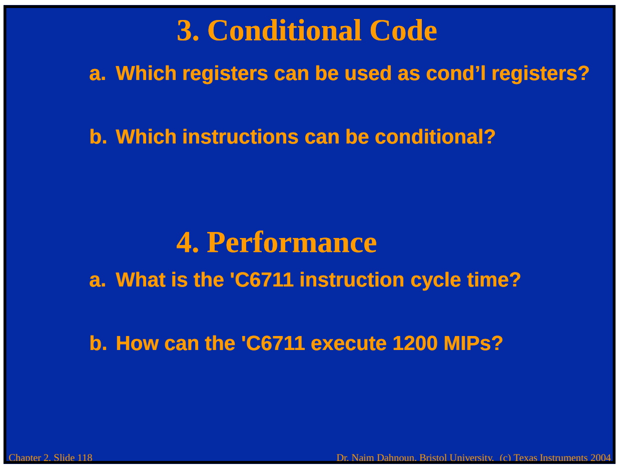
TMS320	= TI DSP
L	= Place holder for voltage levels
C6	= C6x family
2	= Fixed-point core
01	= Memory/peripheral configuration
PKG	= Pkg designator (actual letters TBD)
A	= -40 to 85C (blank for 0 to 70C)
200	= Core CPU speed in Mhz



Module 1 Exam

1. Functional Units
   a. How many can perform an ADD? Name them.

   b. Which support memory loads/stores?
   .M .S .D .L
2. Memory Map
   a. How many external ranges exist on 'C6201?



3. Conditional Code

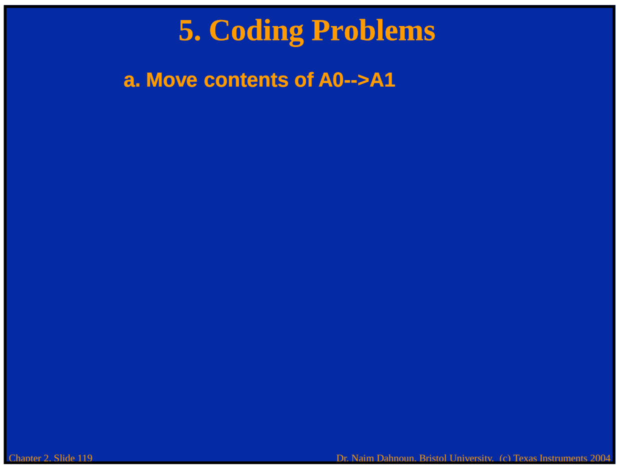
a. Which registers can be used as cond'l registers?

b. Which instructions can be conditional?

4. Performance

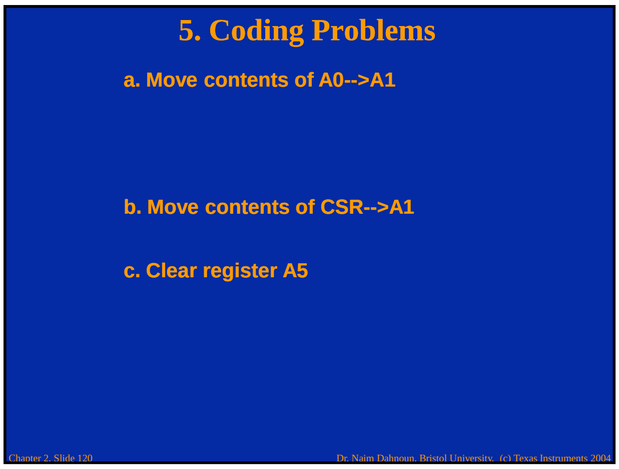
a. What is the 'C6711 instruction cycle time?

b. How can the 'C6711 execute 1200 MIPs?



5. Coding Problems

a. Move contents of A0-->A1

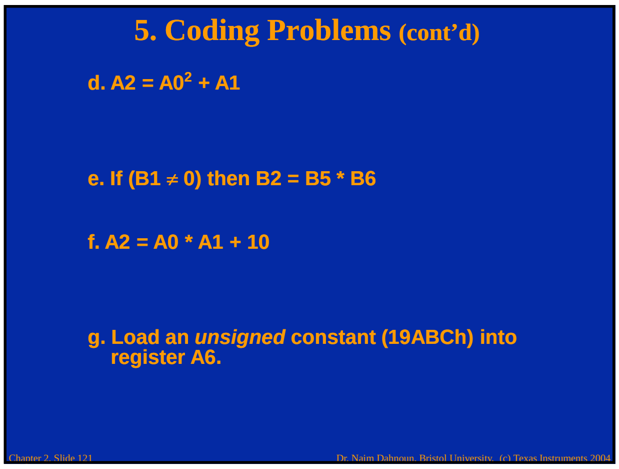


5. Coding Problems

a. Move contents of A0-->A1

b. Move contents of CSR-->A1

c. Clear register A5



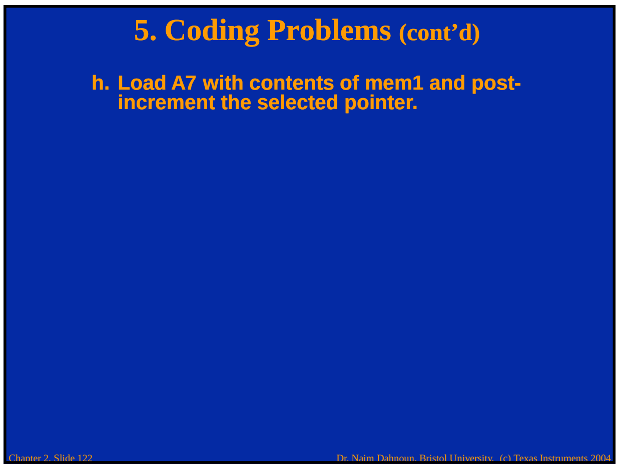
5. Coding Problems (cont'd)

d. $A2 = A0^2 + A1$

e. If ($B1 \neq 0$) then $B2 = B5 * B6$

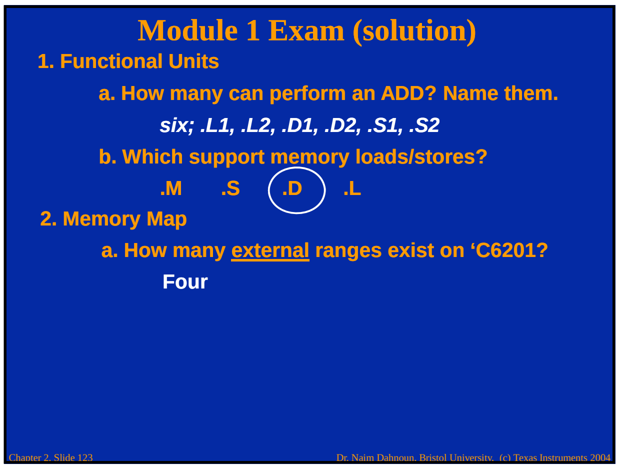
f. $A2 = A0 * A1 + 10$

g. Load an *unsigned* constant (19ABCh) into register A6.



5. Coding Problems (cont'd)

h. Load A7 with contents of mem1 and post-increment the selected pointer.



Module 1 Exam (solution)

1. Functional Units

a. How many can perform an ADD? Name them.

six; .L1, .L2, .D1, .D2, .S1, .S2

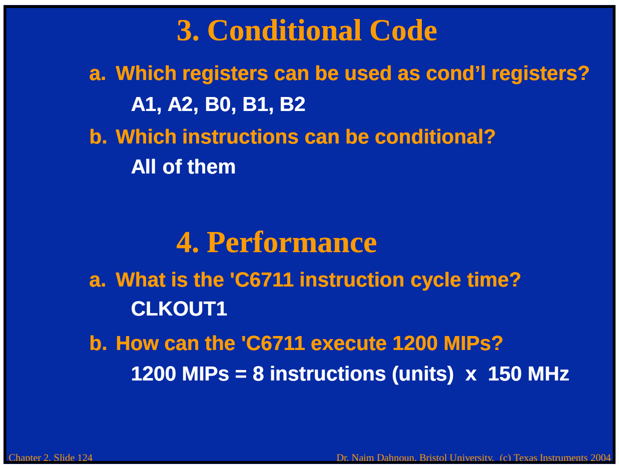
b. Which support memory loads/stores?

.M .S [✓ .D] .L

2. Memory Map

a. How many <u>external</u> ranges exist on 'C6201?

Four



3. Conditional Code

a. Which registers can be used as cond'l registers?

A1, A2, B0, B1, B2

b. Which instructions can be conditional?

All of them

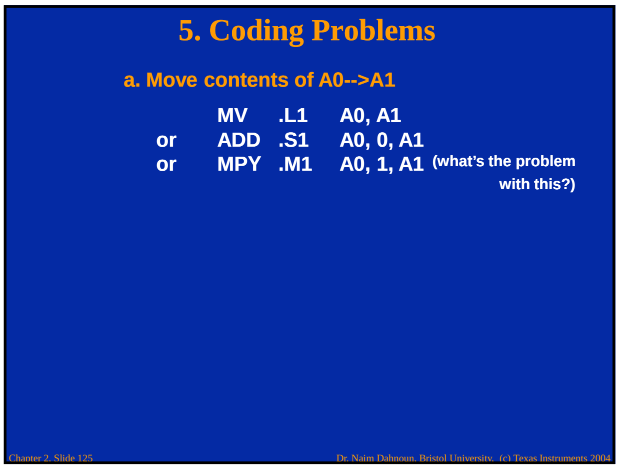
4. Performance

a. What is the 'C6711 instruction cycle time?

CLKOUT1

b. How can the 'C6711 execute 1200 MIPs?

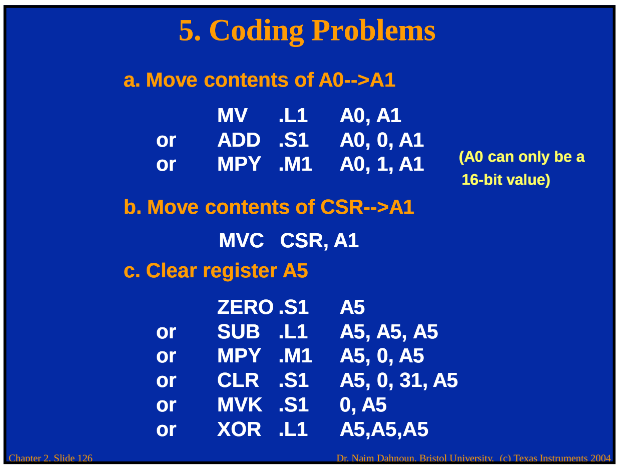
1200 MIPs = 8 instructions (units) x 150 MHz



5. Coding Problems

a. Move contents of A0-->A1

```
        MV    .L1    A0, A1
or      ADD   .S1    A0, 0, A1
or      MPY   .M1    A0, 1, A1  (what's the problem
                                          with this?)
```



5. Coding Problems

a. Move contents of A0-->A1

```
        MV    .L1    A0, A1
or      ADD   .S1    A0, 0, A1
or      MPY   .M1    A0, 1, A1
```

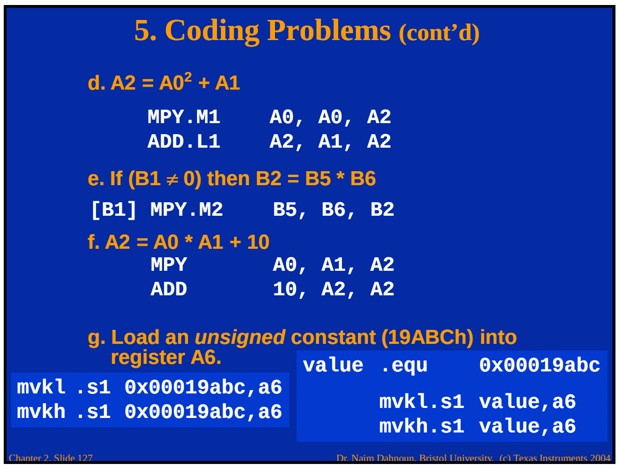
(A0 can only be a 16-bit value)

b. Move contents of CSR-->A1

```
        MVC   CSR, A1
```

c. Clear register A5

```
        ZERO .S1    A5
or      SUB   .L1    A5, A5, A5
or      MPY   .M1    A5, 0, A5
or      CLR   .S1    A5, 0, 31, A5
or      MVK   .S1    0, A5
or      XOR   .L1    A5,A5,A5
```



5. Coding Problems (cont'd)

d. $A2 = A0^2 + A1$

```
MPY.M1     A0, A0, A2
ADD.L1     A2, A1, A2
```

e. If ($B1 \neq 0$) then B2 = B5 * B6

```
[B1] MPY.M2     B5, B6, B2
```

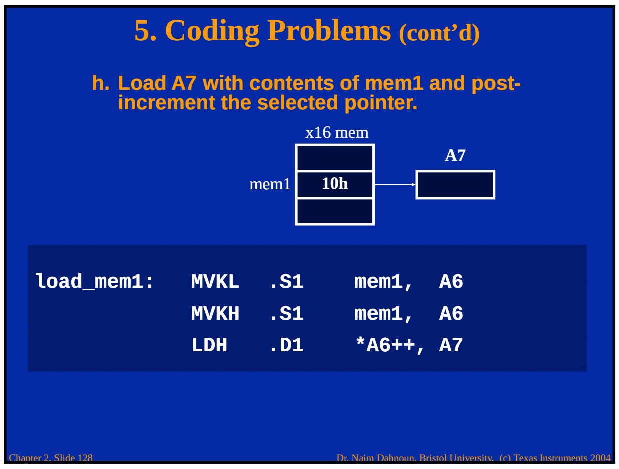
f. A2 = A0 * A1 + 10

```
MPY        A0, A1, A2
ADD        10, A2, A2
```

g. Load an *unsigned* constant (19ABCh) into register A6.

```
mvkl .s1 0x00019abc,a6
mvkh .s1 0x00019abc,a6
```

```
value  .equu    0x00019abc

       mvkl.s1  value,a6
       mvkh.s1  value,a6
```

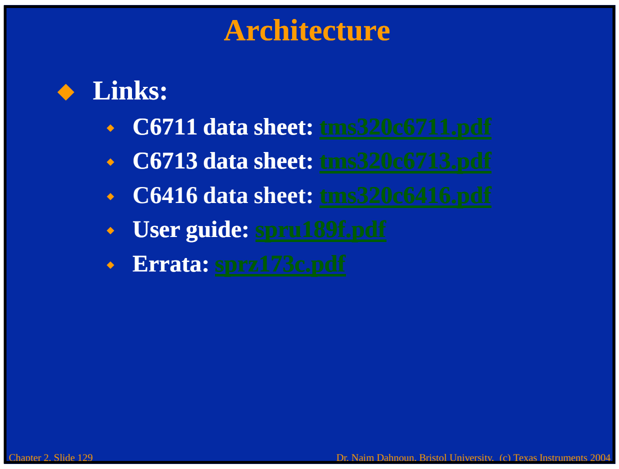


5. Coding Problems (cont'd)

h. Load A7 with contents of mem1 and post-increment the selected pointer.

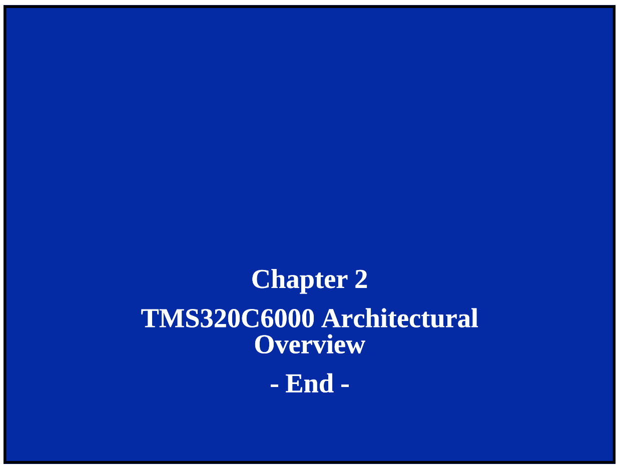
x16 mem
A7
mem1
10h

```
load_mem1:   MVKL   .S1     mem1,   A6
             MVKH   .S1     mem1,   A6
             LDH    .D1     *A6++,  A7
```



Architecture

- **Links:**
  - **C6711 data sheet:** tms320c6711.pdf
  - **C6713 data sheet:** tms320c6713.pdf
  - **C6416 data sheet:** tms320c6416.pdf
  - **User guide:** spru189f.pdf
  - **Errata:** sprz173c.pdf



Chapter 2

TMS320C6000 Architectural Overview

- End -