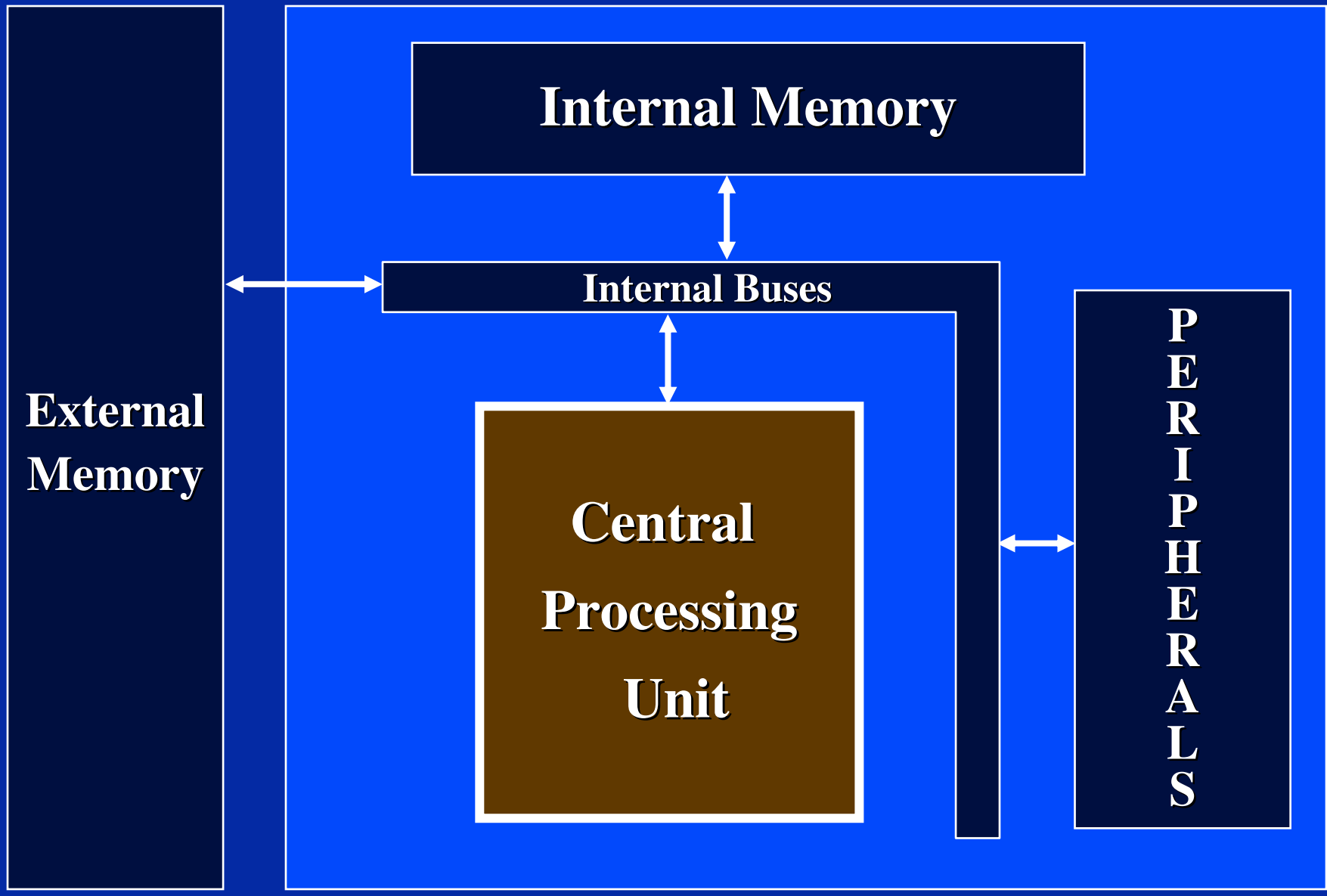
TMS320C6000 Architectural Overview

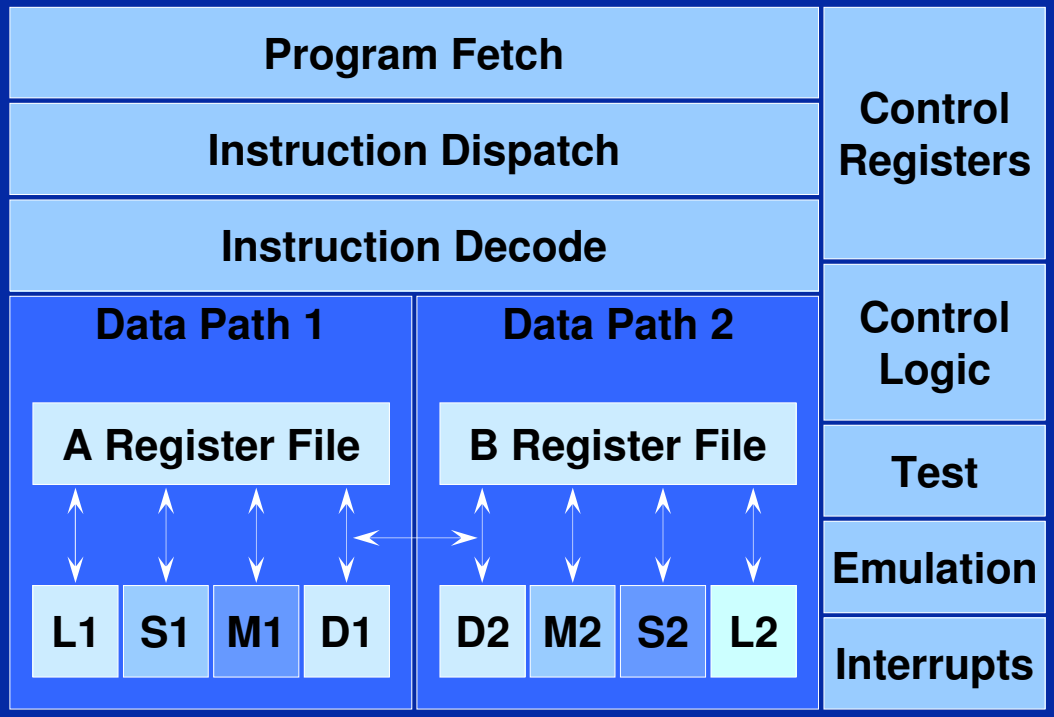
Learning Objectives

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

General DSP System Block Diagram



TMS320C6000 DSP Cores

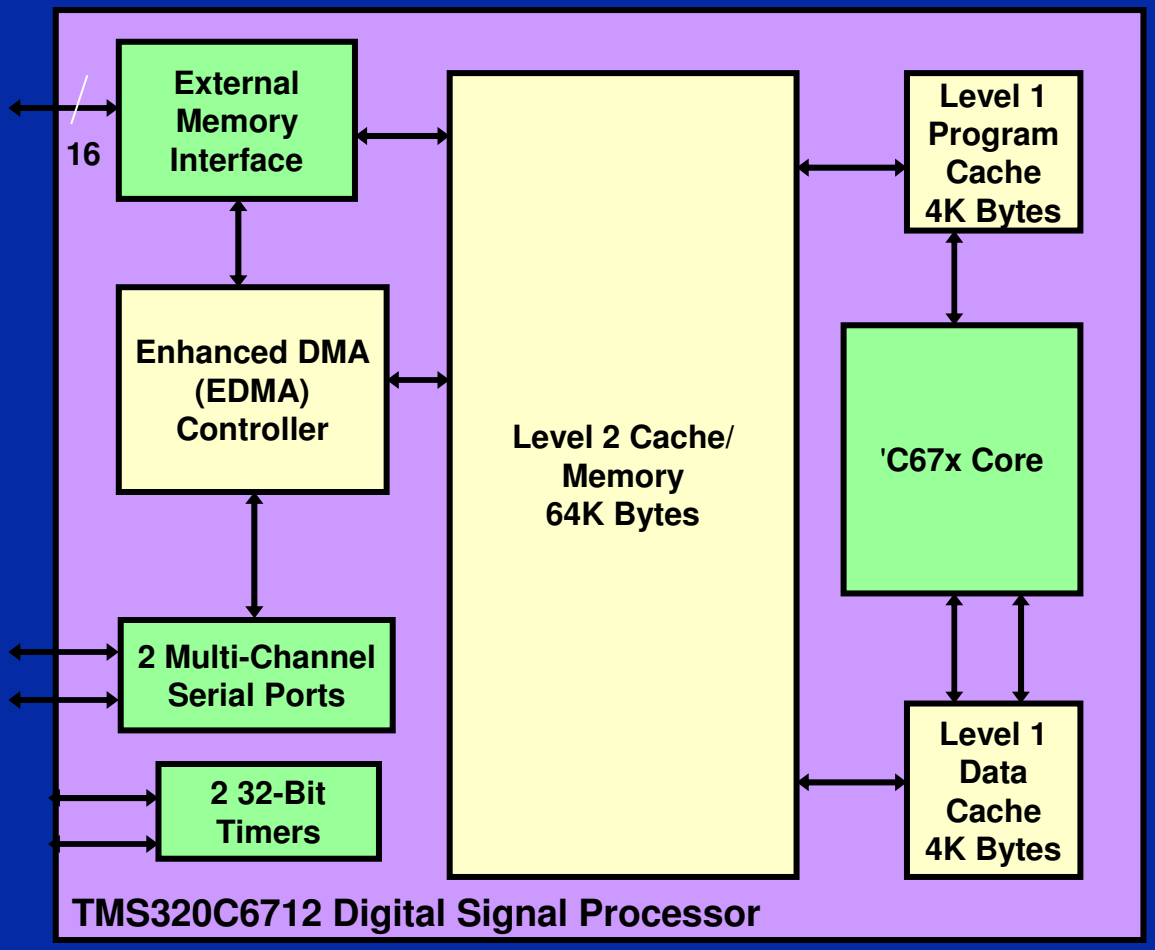


Functional Units

C6000 Functional Units			
L	**S**	**D**	**M**
Integer Adder Logical Integer Comparison Bit Counting	Integer Adder Logical Shifting Bit Manipulation Constant Branch/Control Paired Short Math	Integer Adder Load-Store	Integer Multiplier

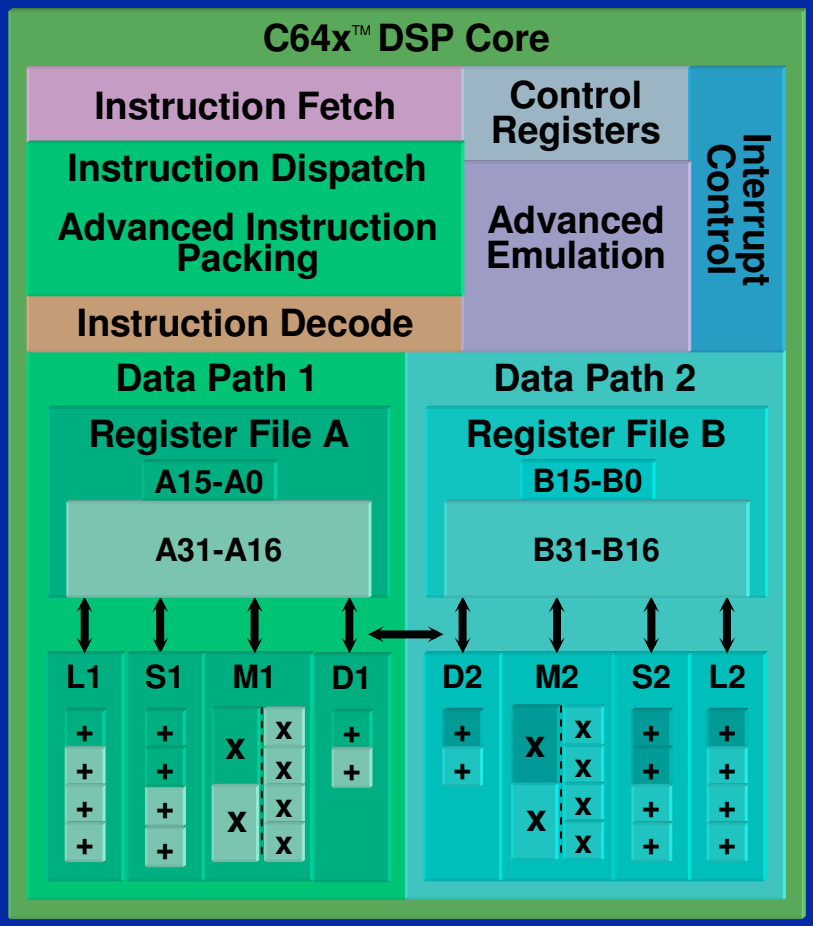
C67x Additions			
L	**S**	**D**	**M**
FP Adder FP Conversion	FP Compare FP Conversion FP Reciprocal	Load Double	FP Multiplier

TMS320C6712



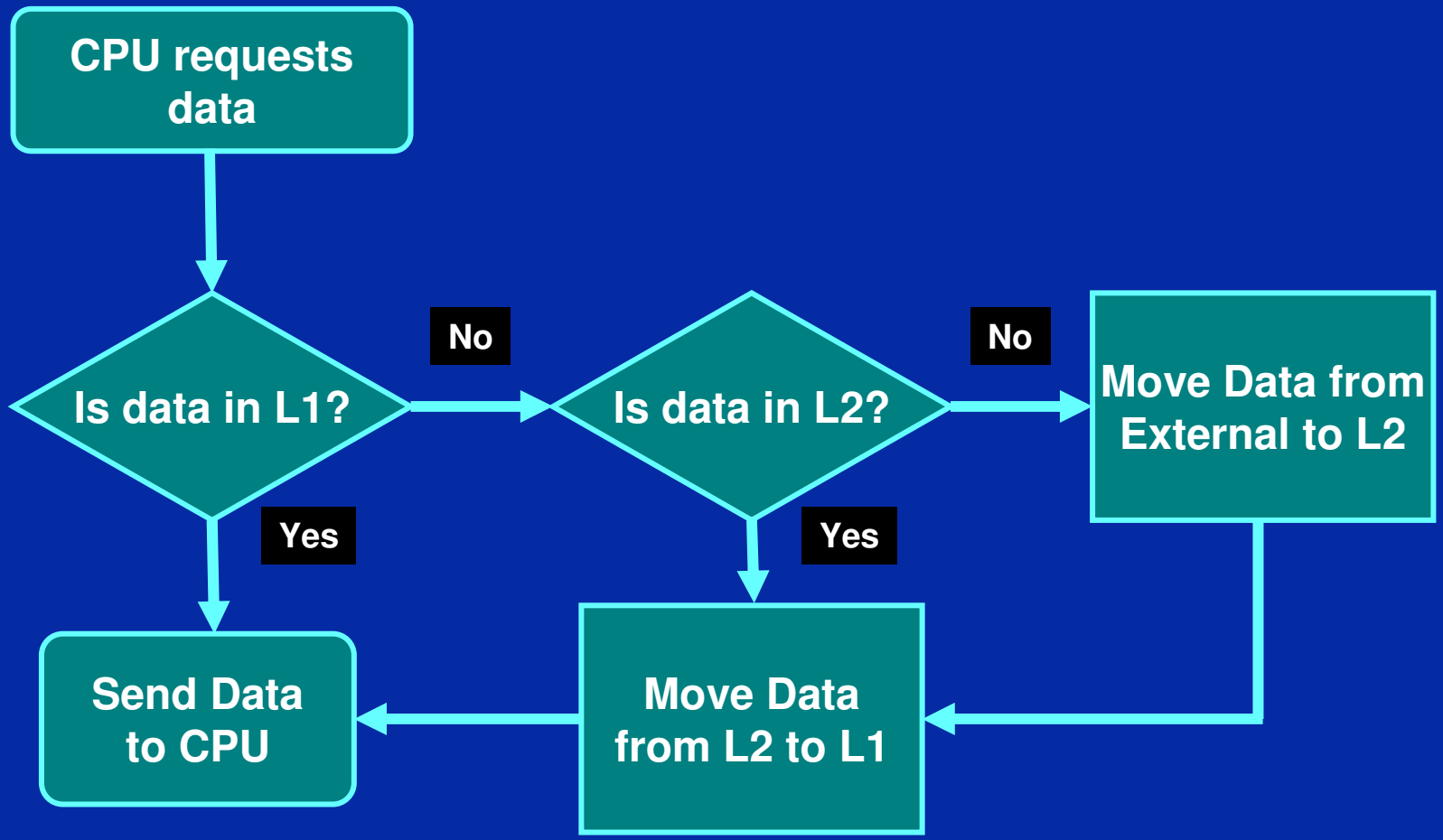
- 600 MFLOPS/100 MHz at $9.95 in Volume
- Code compatibility with all C6000 devices
- Dual-level cache memory architecture (same as C6211/C6711) enables systems savings
- Enhanced DMA (EDMA) is optimized for efficiency in small RAM devices
- 256-pin BGA (same package as C6711)

TMS320C64x™ DSP Achieves Performance Breakthrough



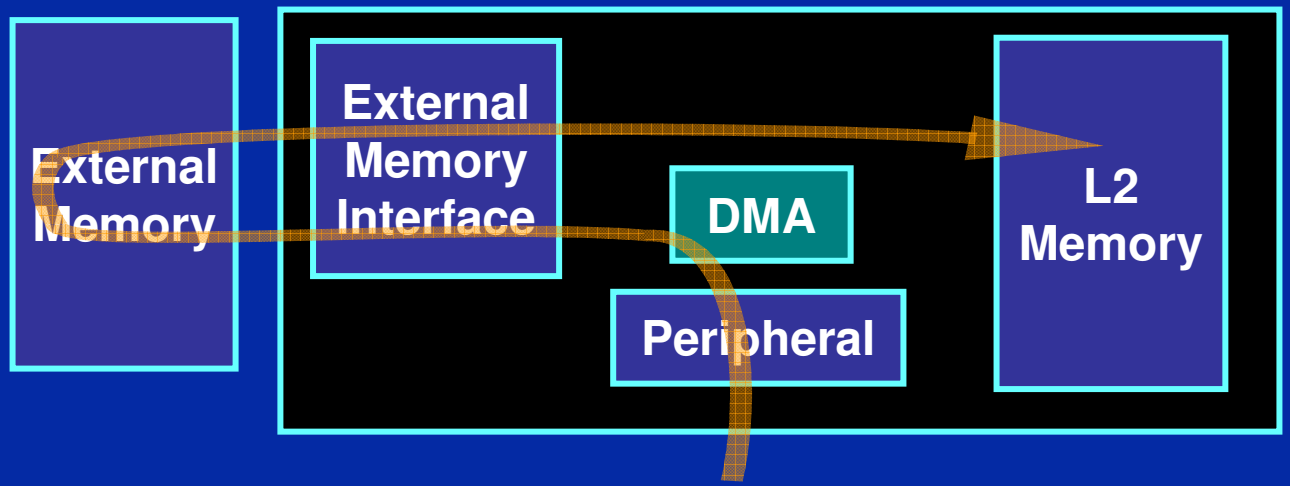
- **Parallelism**
  - 8 new instructions can always be dispatched every cycle
  - Packed data processing within instructions (dual 16-bit, quad 8-bit)
  - Key data flow instructions
- **Special Purpose Instructions**
  - Accelerate key broadband communications and imaging functions

Cache Data Flow



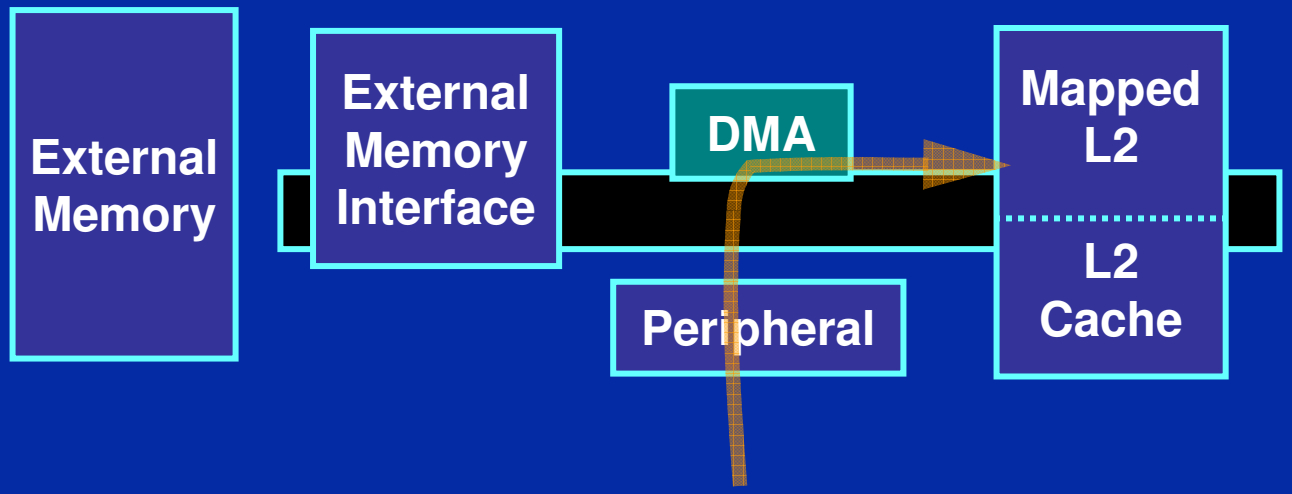
L2: Direct On-Chip I/O Typical Cache Architecture

- Lacks non-cacheable regions
- Requires external storage of peripheral data



C6211/C6711/C6712 L2: Direct On-Chip I/O C6712

- Configurable L2 allows real-time processing



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)

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

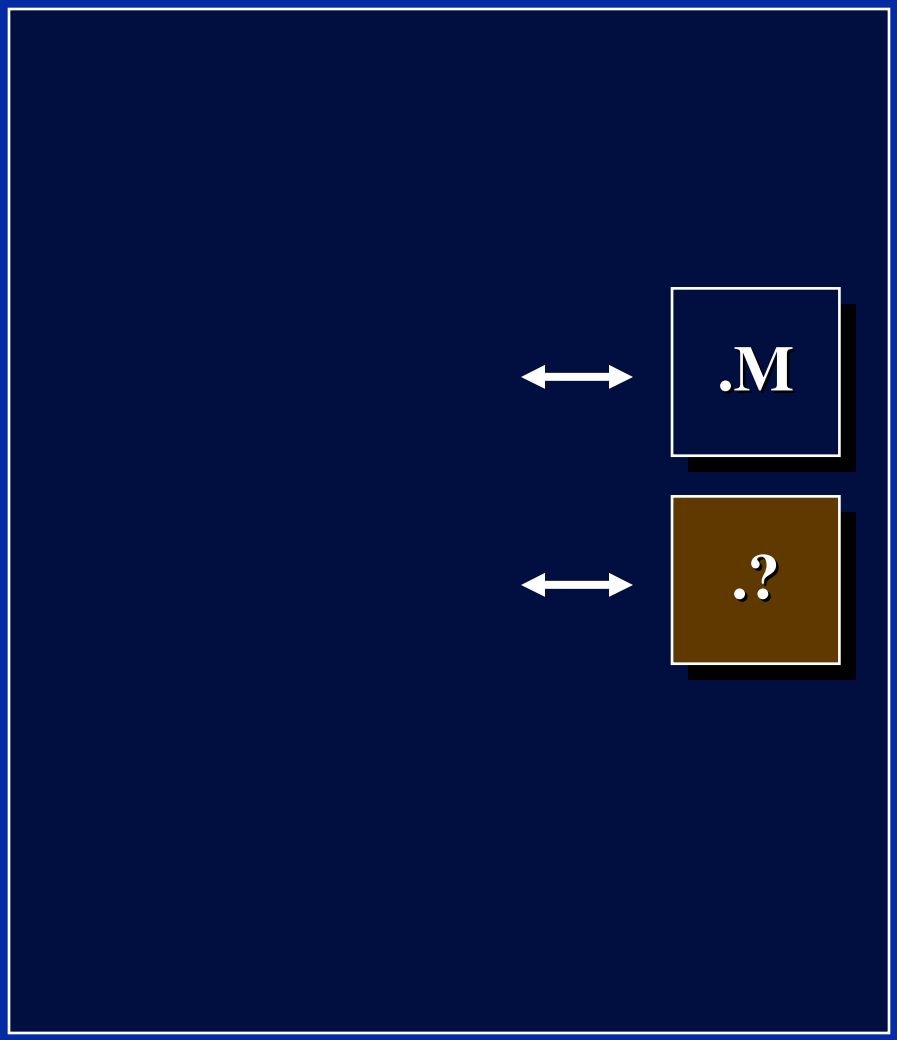
The . M unit performs multiplications in hardware

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.

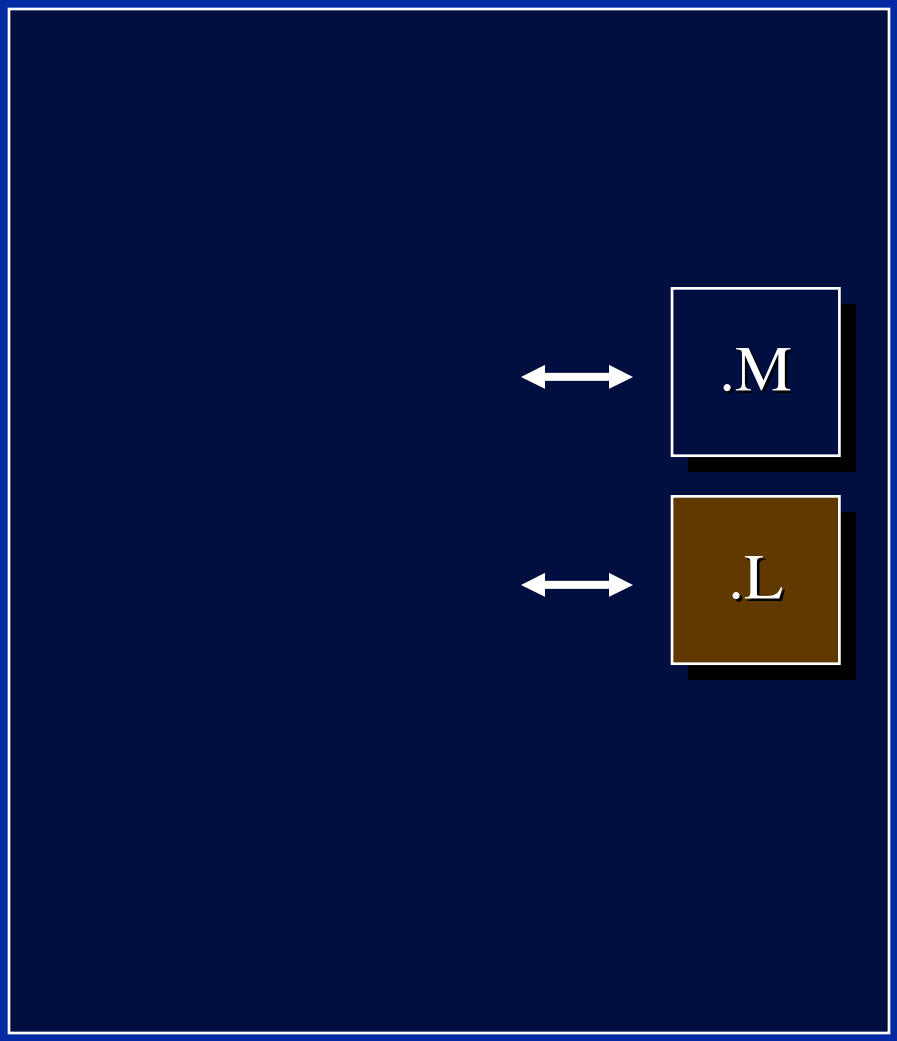
Addition (.?)



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

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

Add (.L unit)

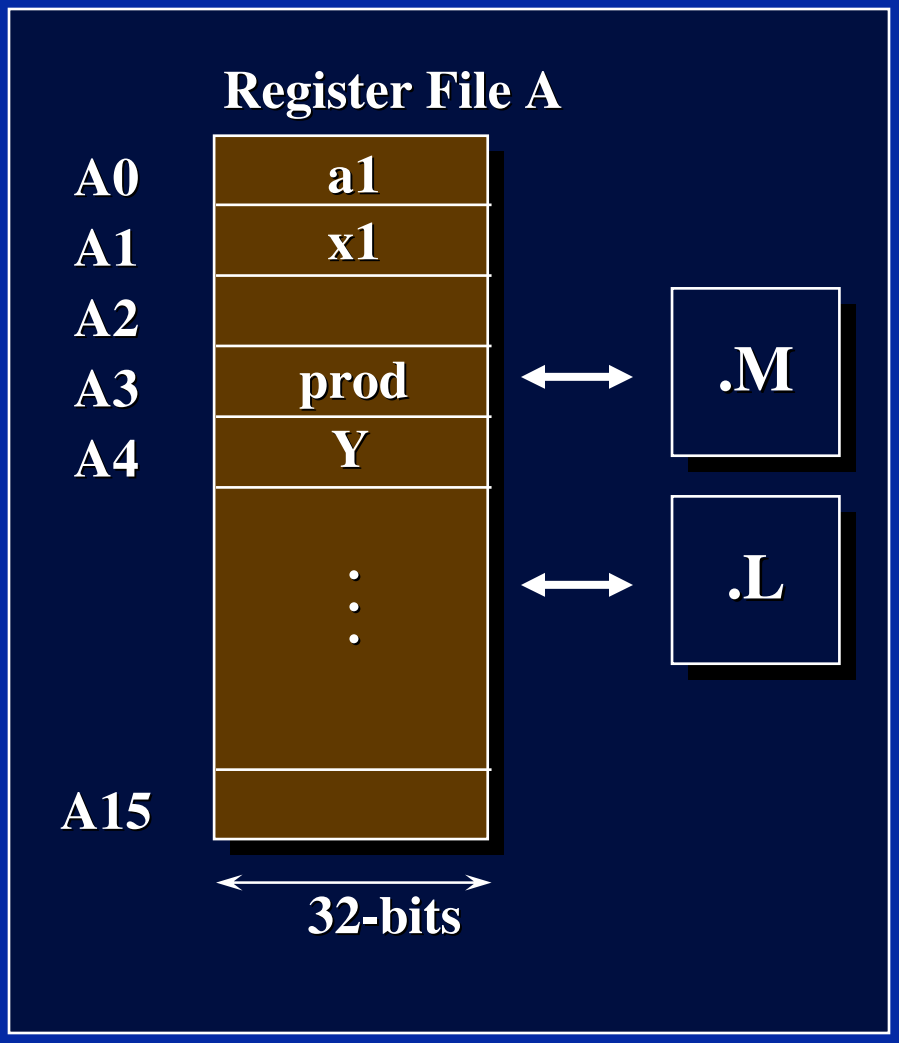


$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
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

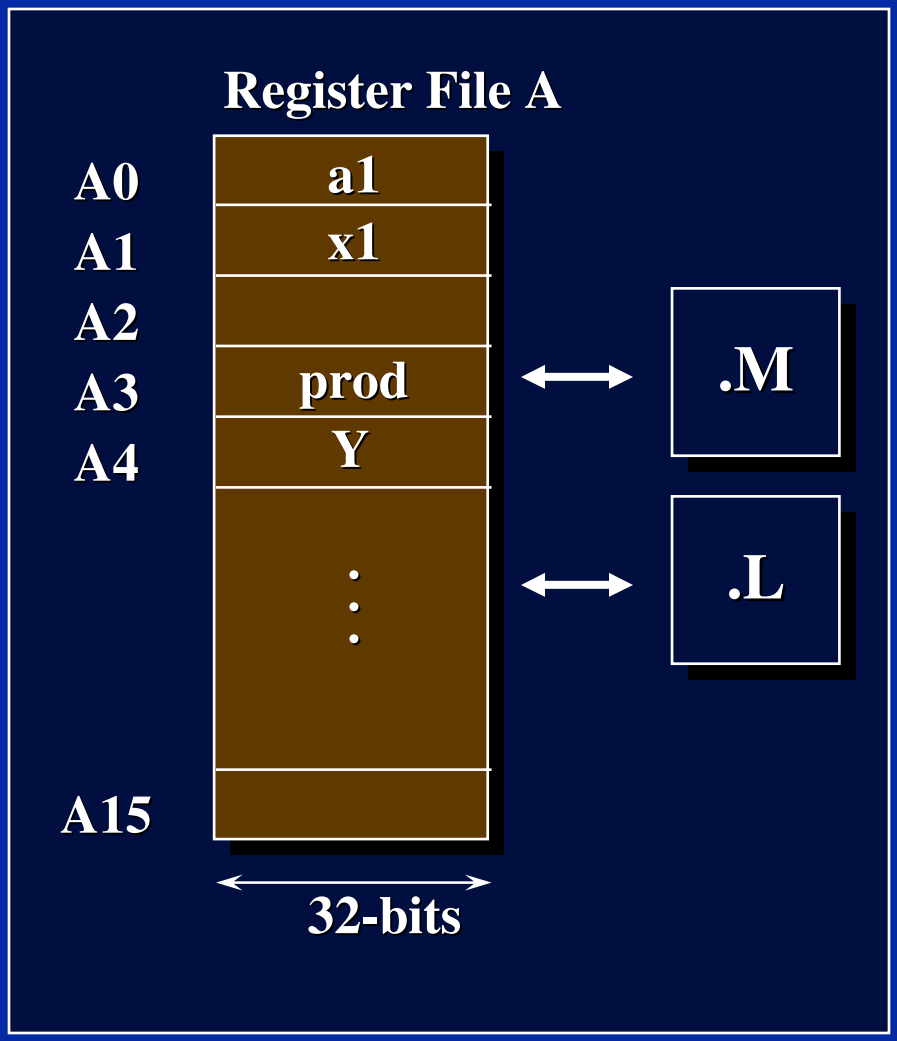


$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
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

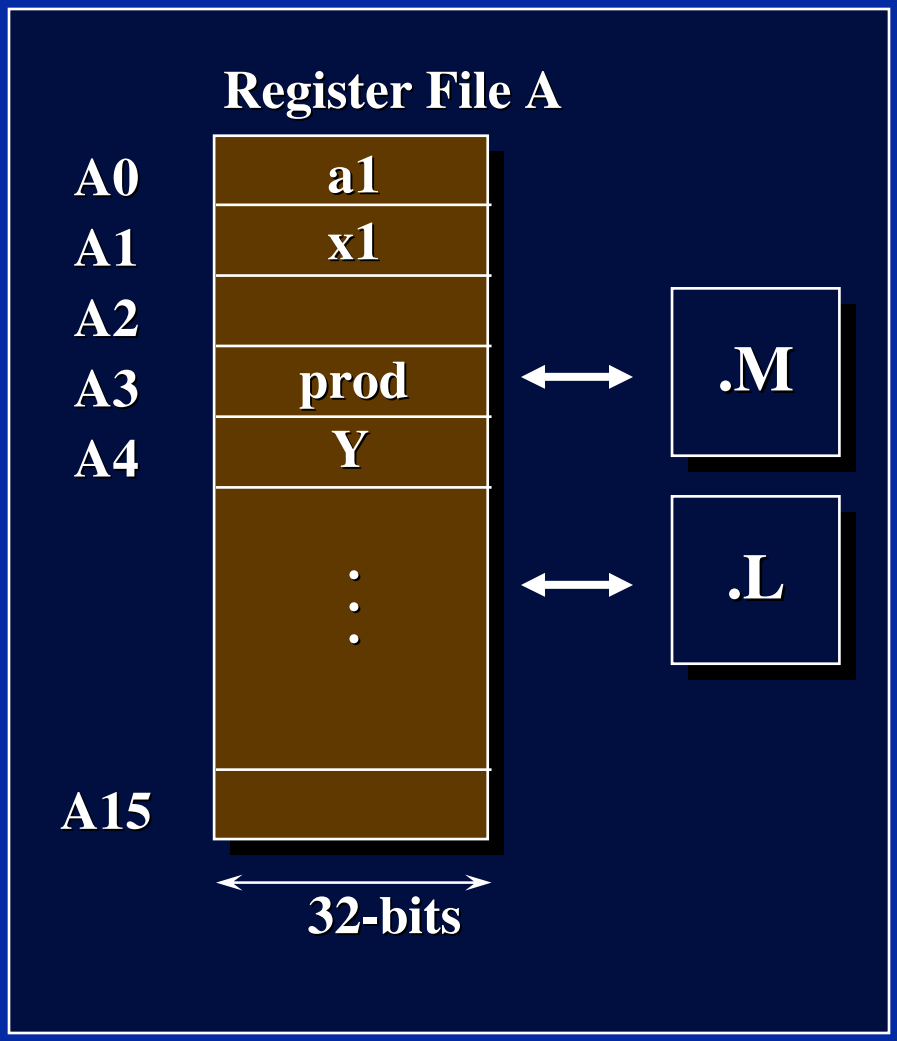


$$Y = \sum_{n=1}^{40} a_n * x_n$$

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

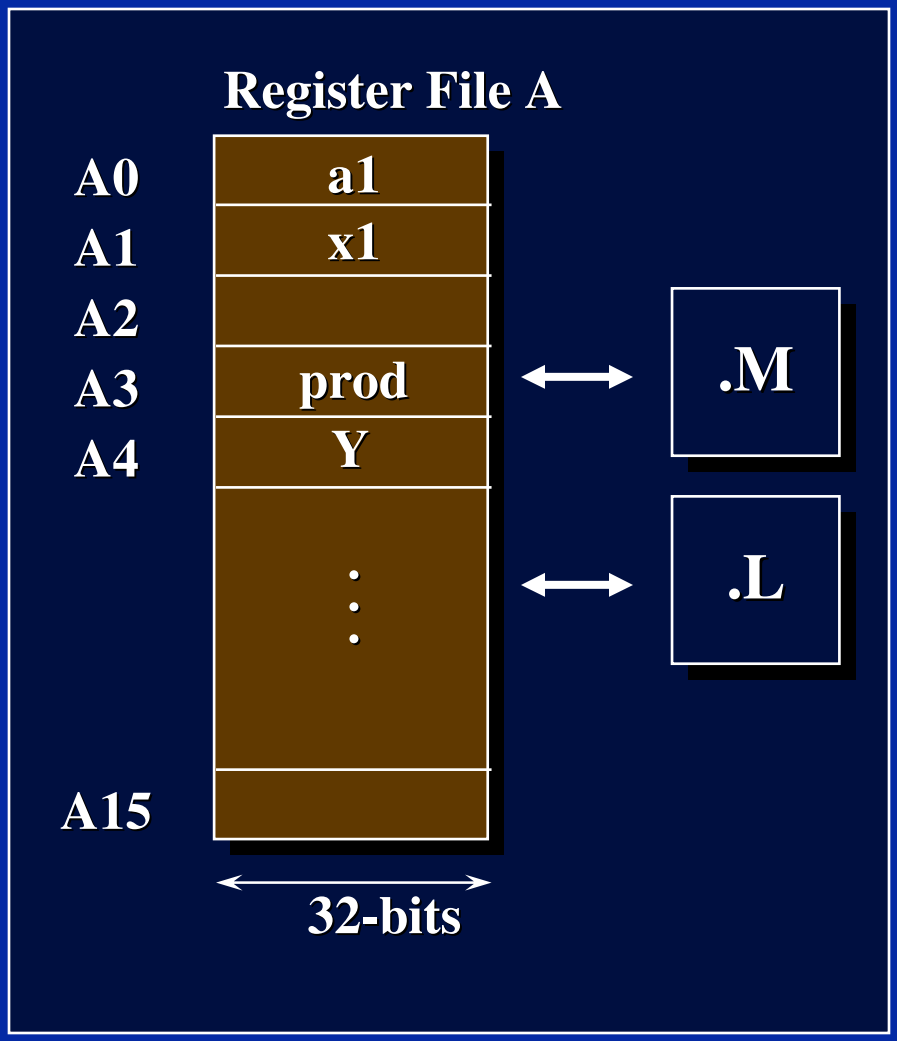


$$Y = \sum_{n=1}^{40} a_n * x_n$$

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

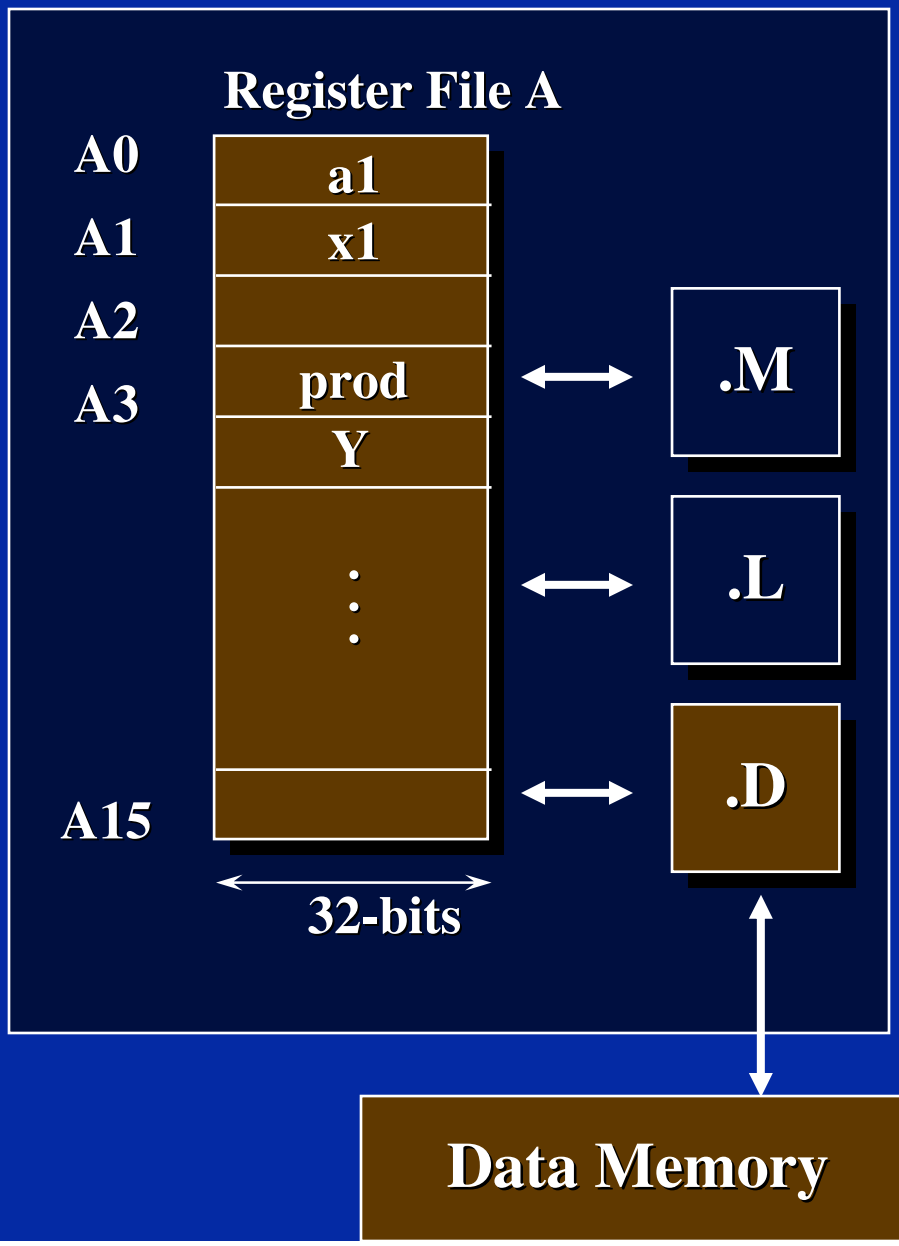
Register File A contains 16 registers (A0 -A15) which are 32-bits wide.

Data loading



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

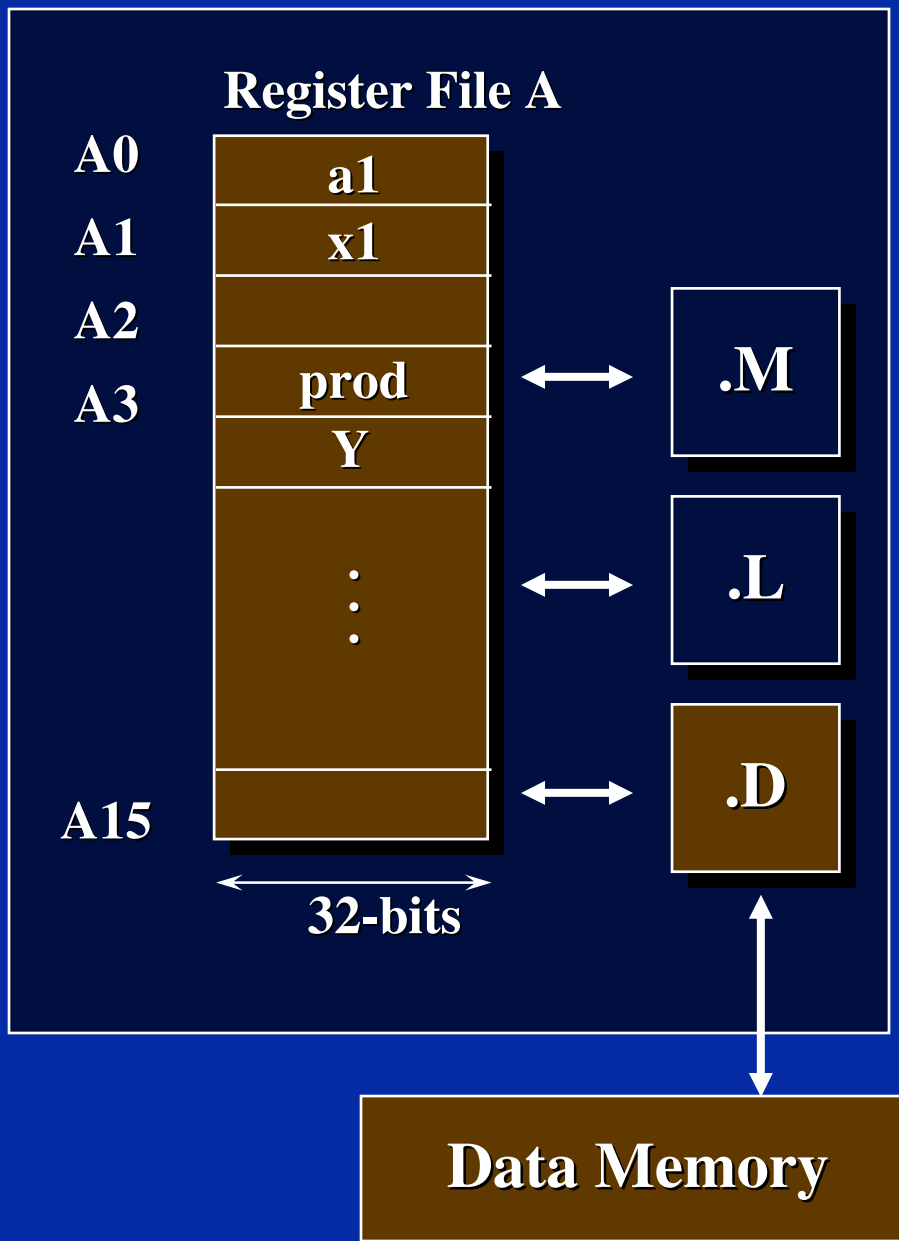
Load Unit ".D"



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

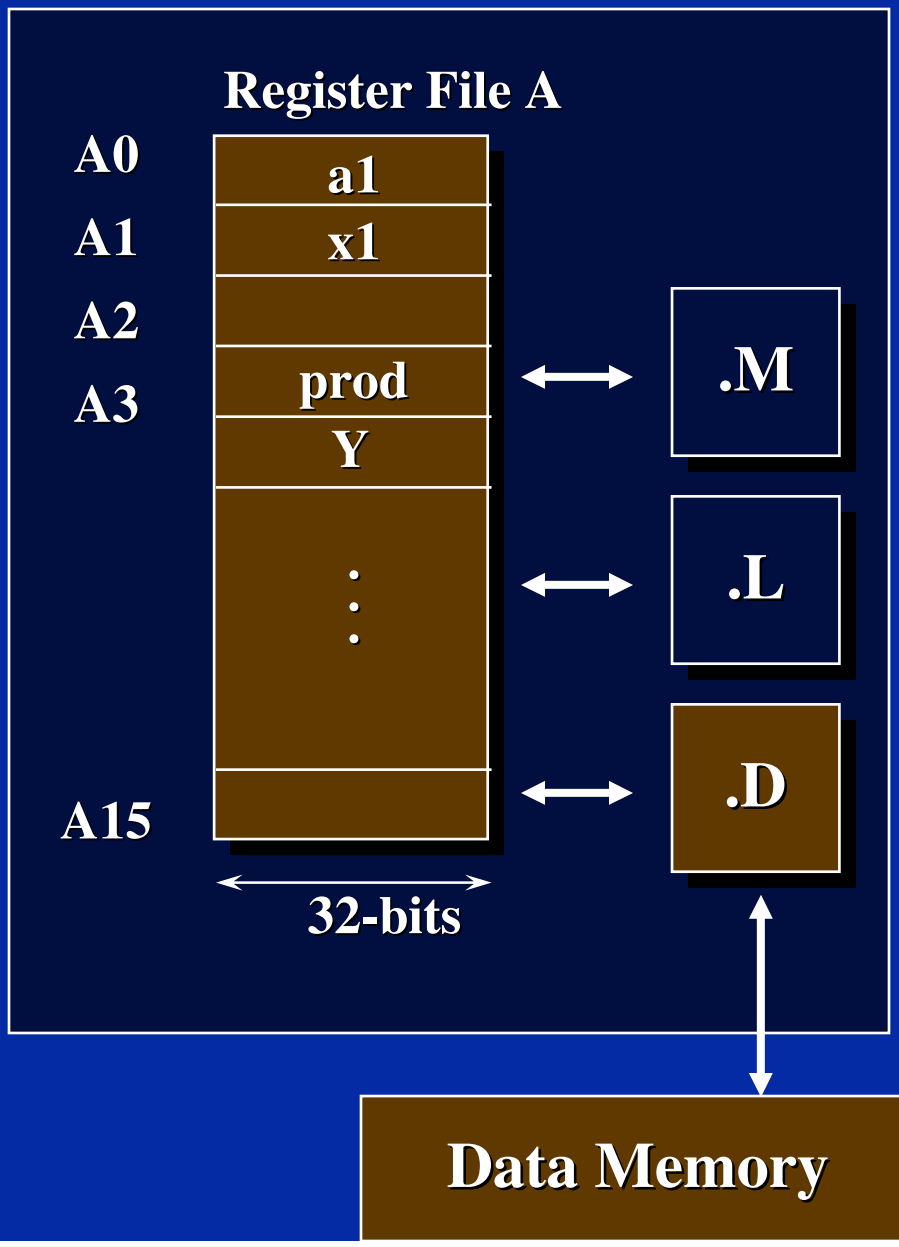
A: The operands are loaded into the registers by loading them from the memory using the .D unit.

Load Unit ".D"



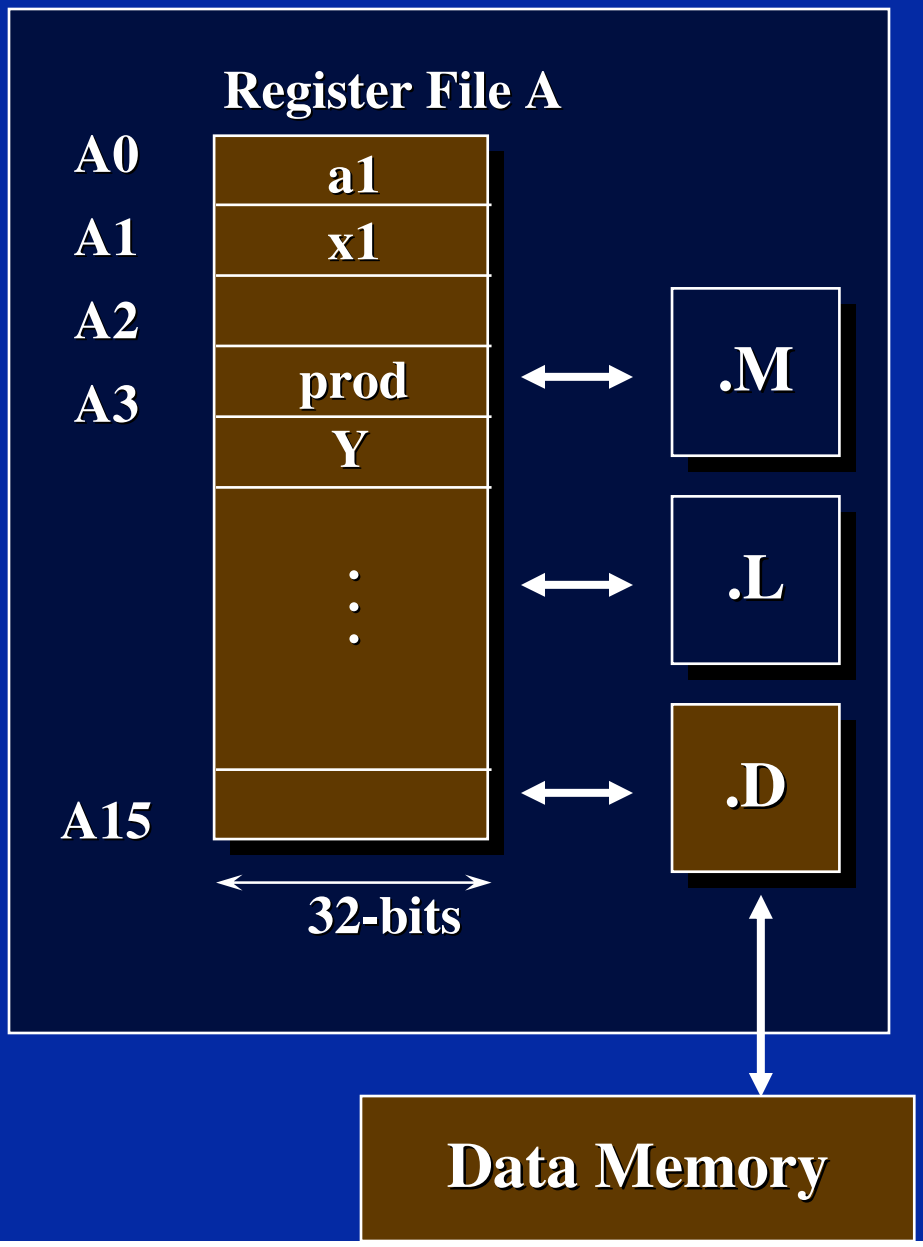
It is worth noting at this stage that the only way to access memory is through the .D unit.

Load Instruction



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

Load Instructions (LDB, LDH,LDW,LDDW)



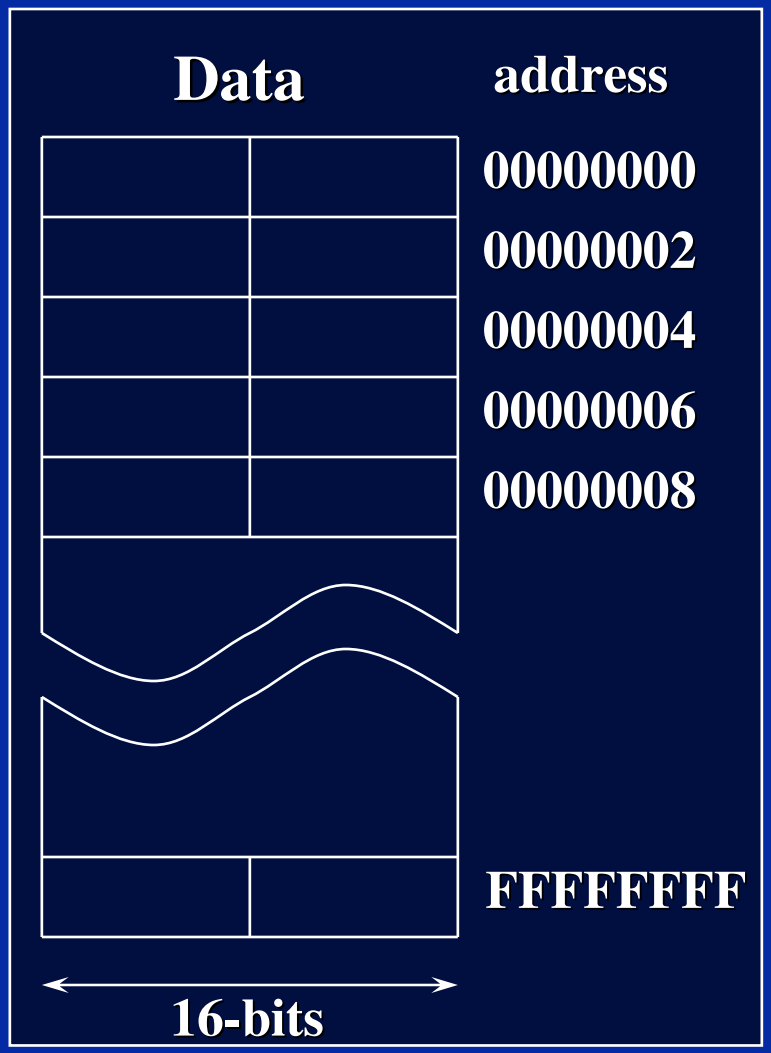
Q: Which instruction(s) can be used for loading operands from the memory to the registers?

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:

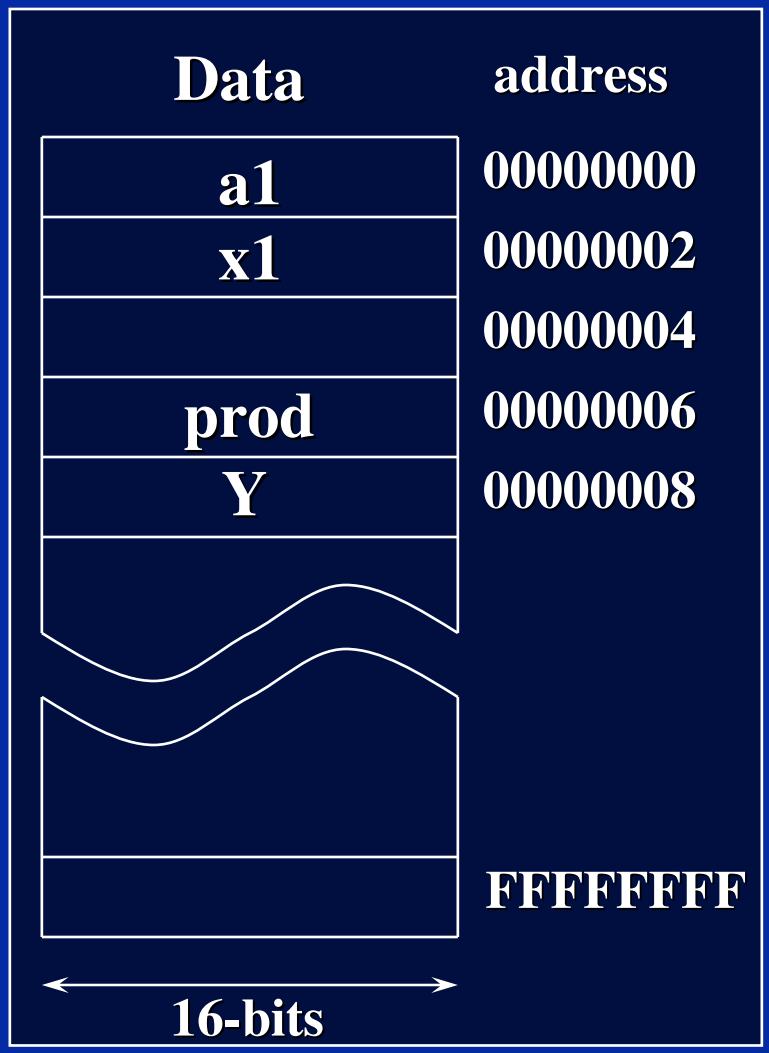
```
LD   *Rn,Rm
```

Where:

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

and

Rm is the destination register.

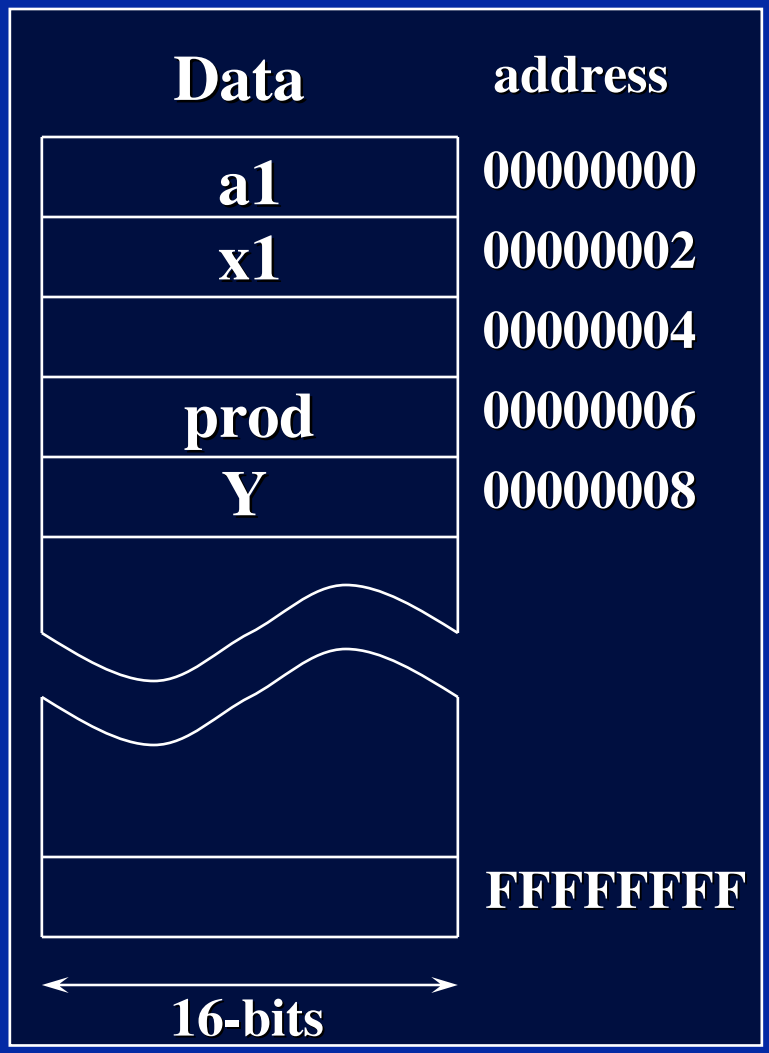


Using the Load Instructions

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?



Using the Load Instructions

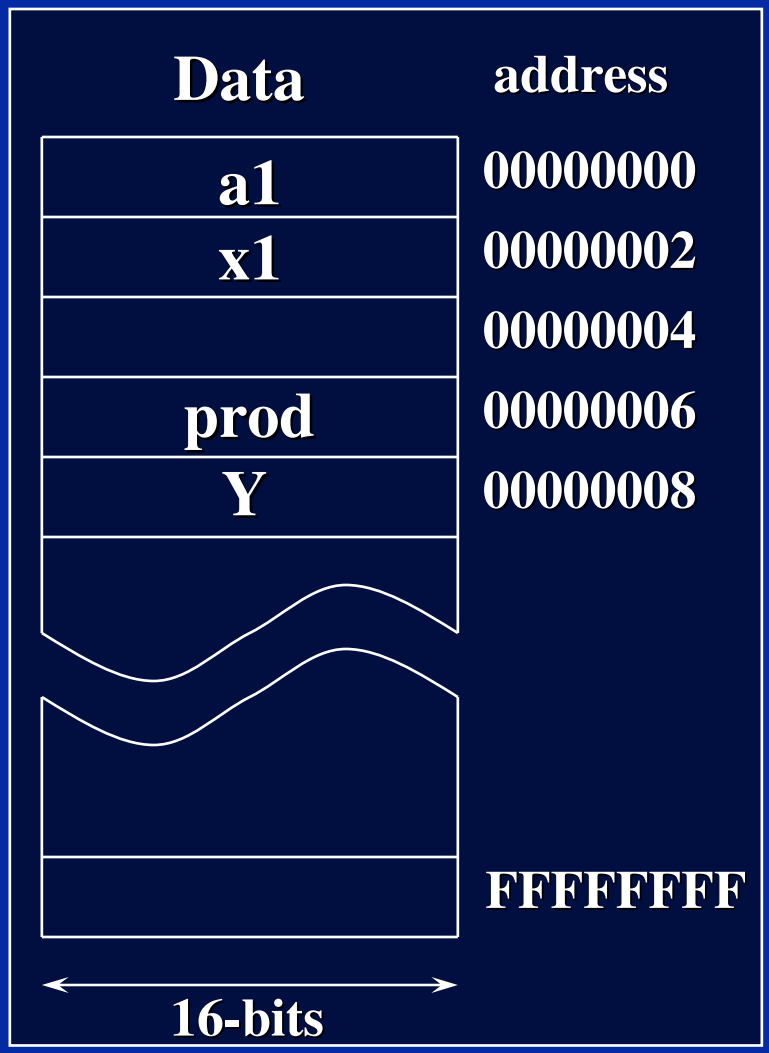
The syntax for the load instruction is:

```
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.



Using the Load Instructions

The syntax for the load instruction is:

```
LD   *Rn,Rm
```

Example:

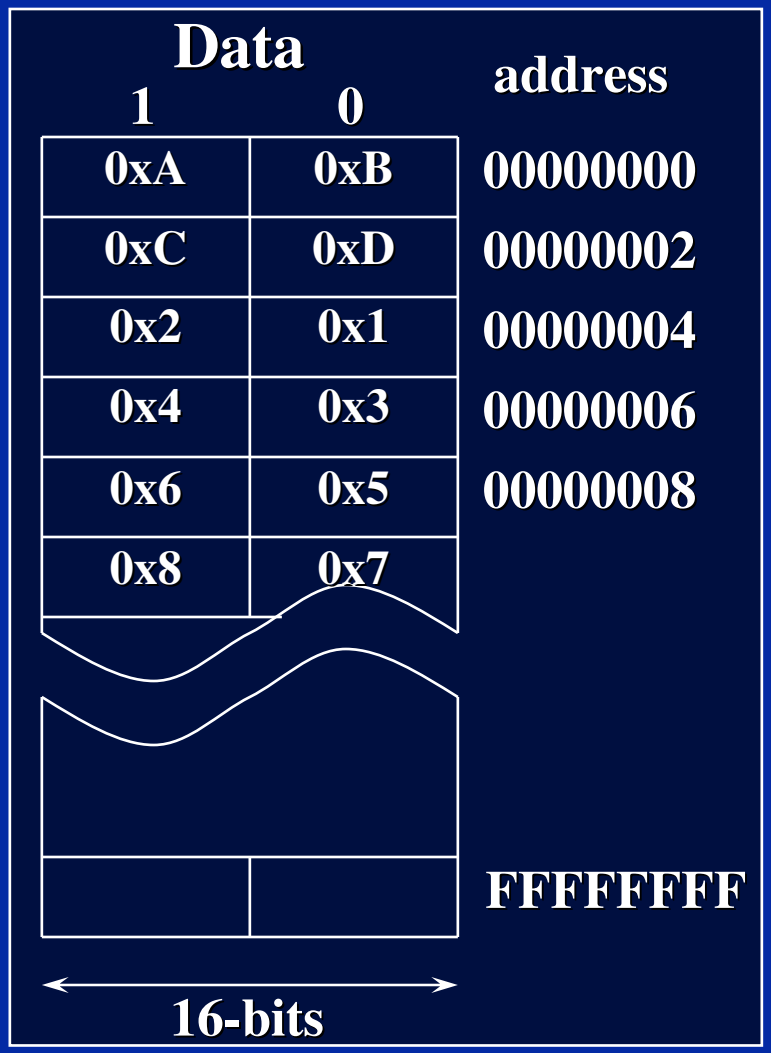
If we assume that A5 = 0x4 then:

(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



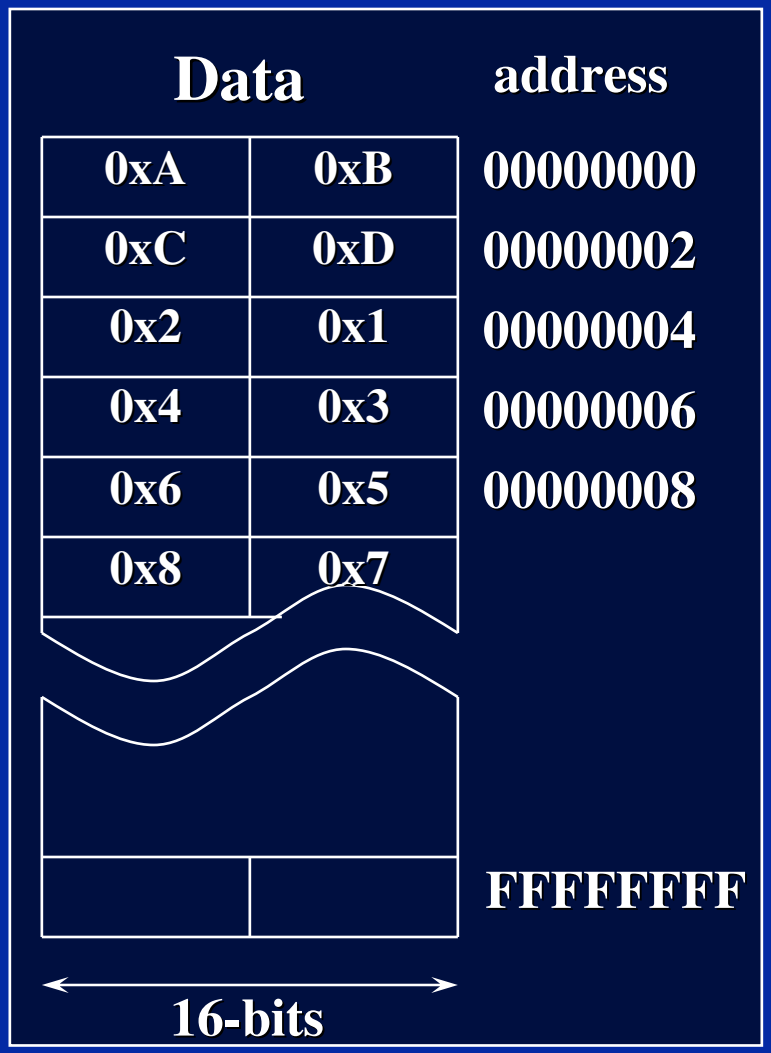
Using the Load Instructions

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?



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).

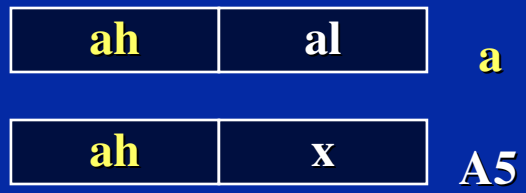
Loading the Pointer Rn

- 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)

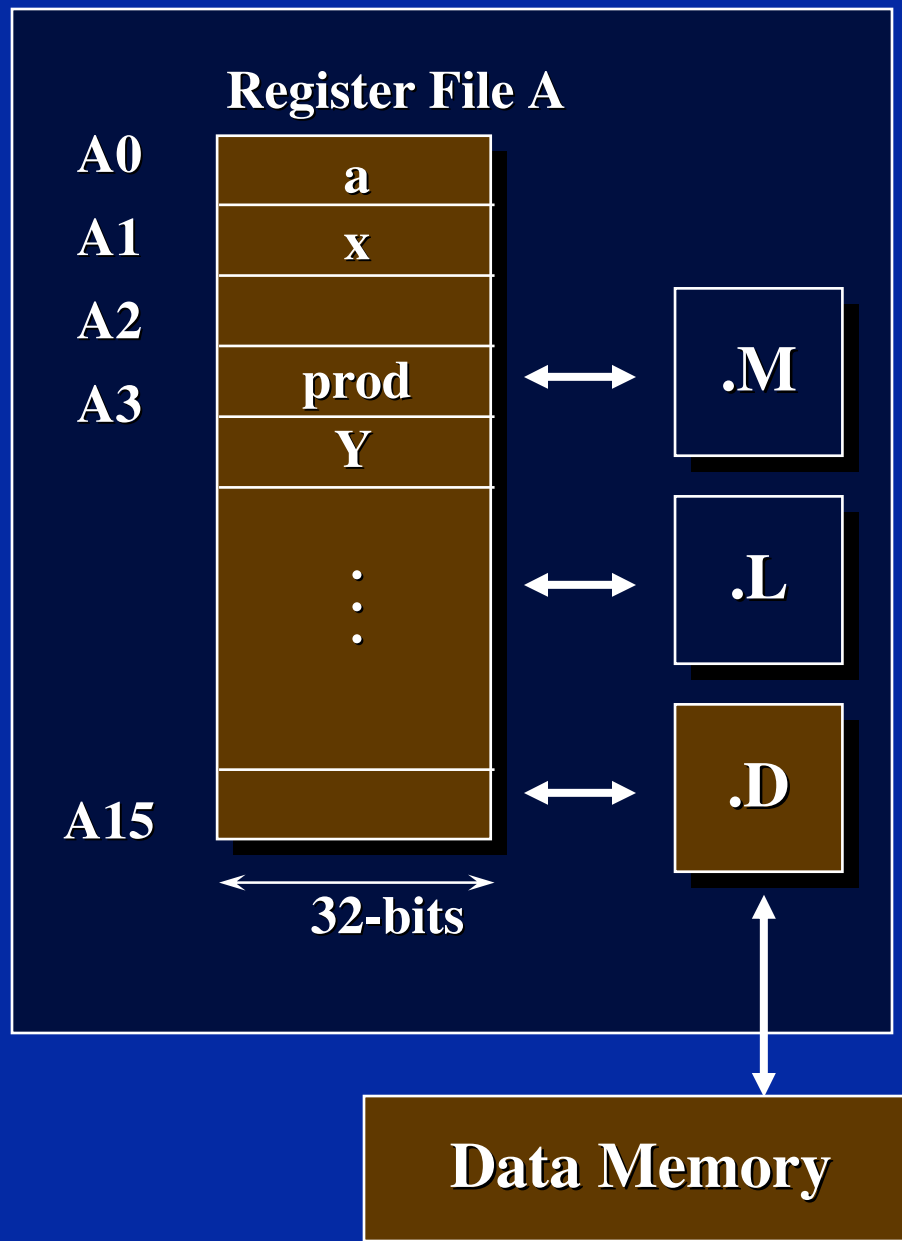
```
MVKH    0x1234FABC, A5
```
A5 = 0x1234FABC ; OK

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
```

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.

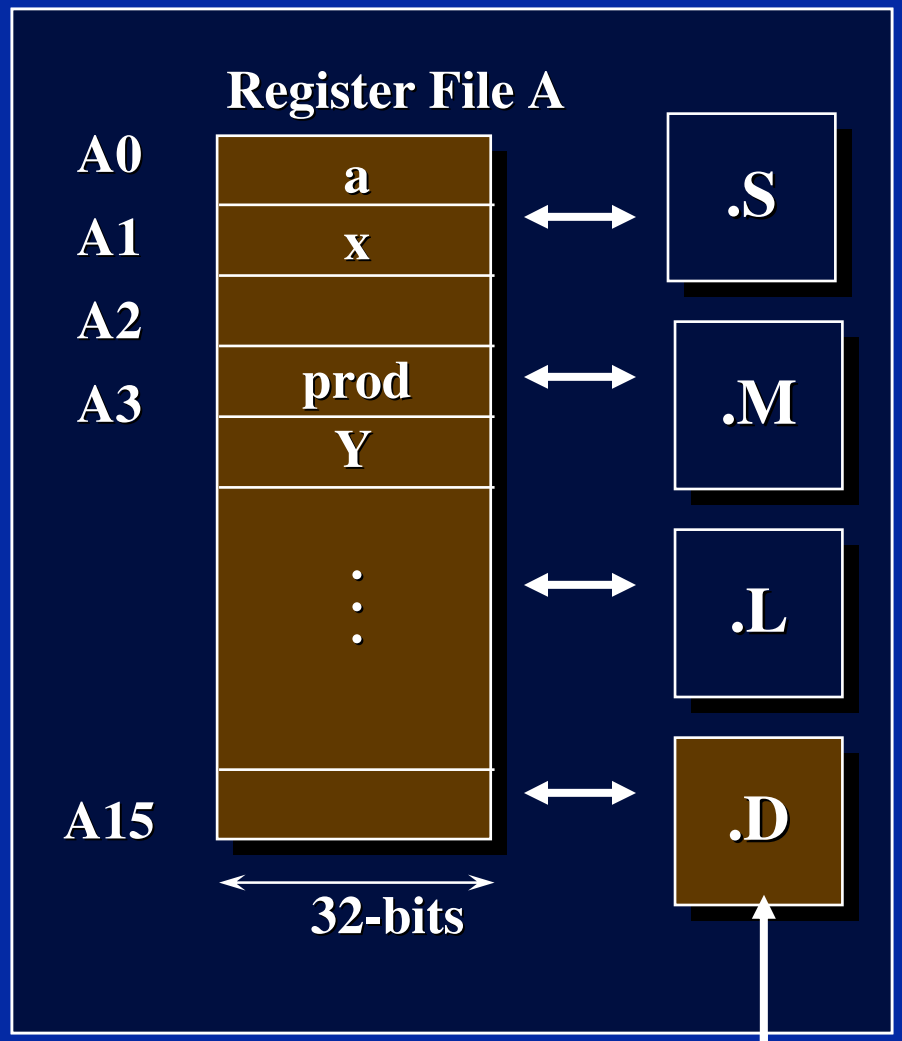
1. Create a label to branch to

```
        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
```

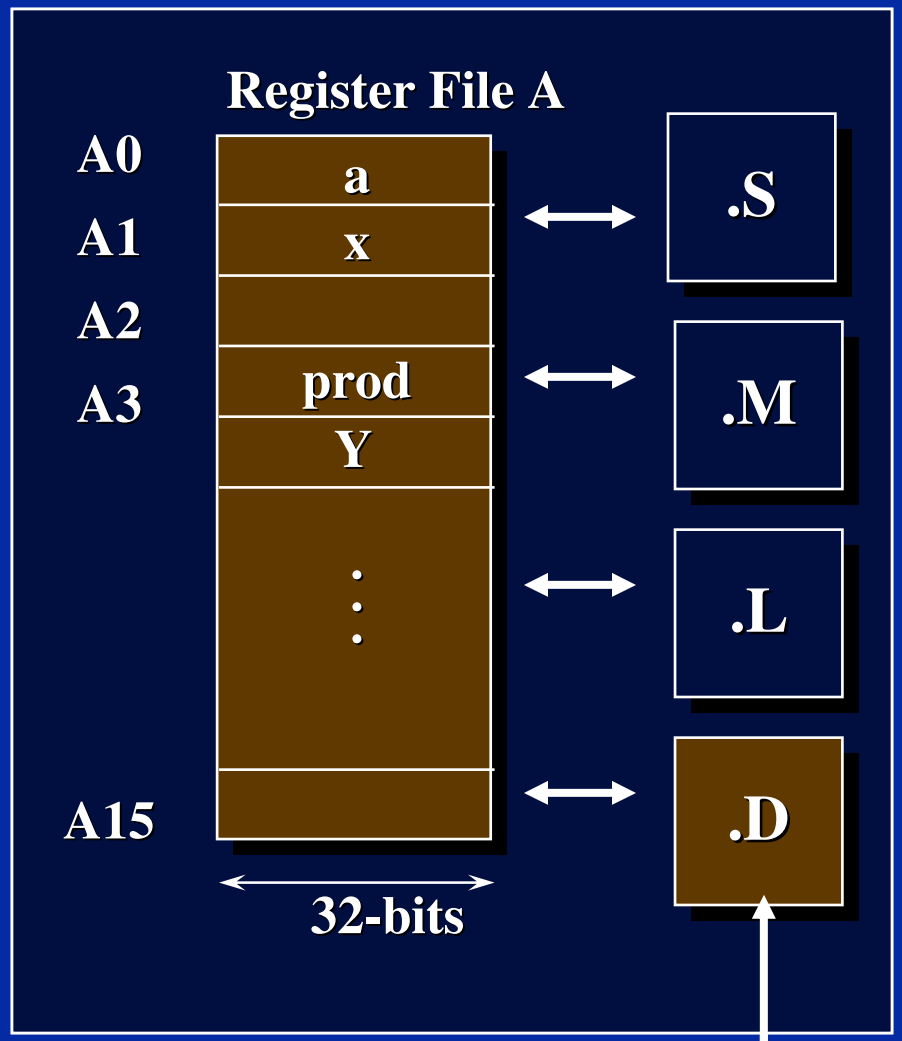
Which unit is used by the B instruction?




```
                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
```

Data Memory

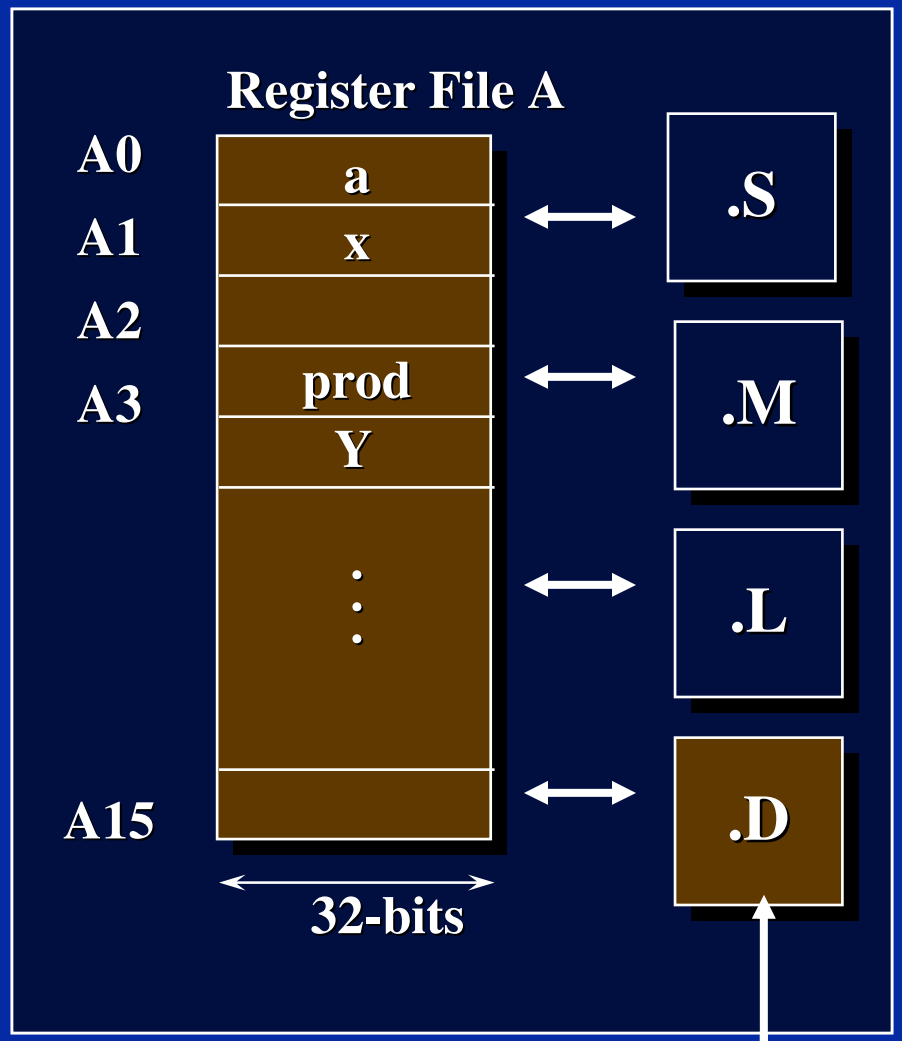
Which unit is used by the B instruction?

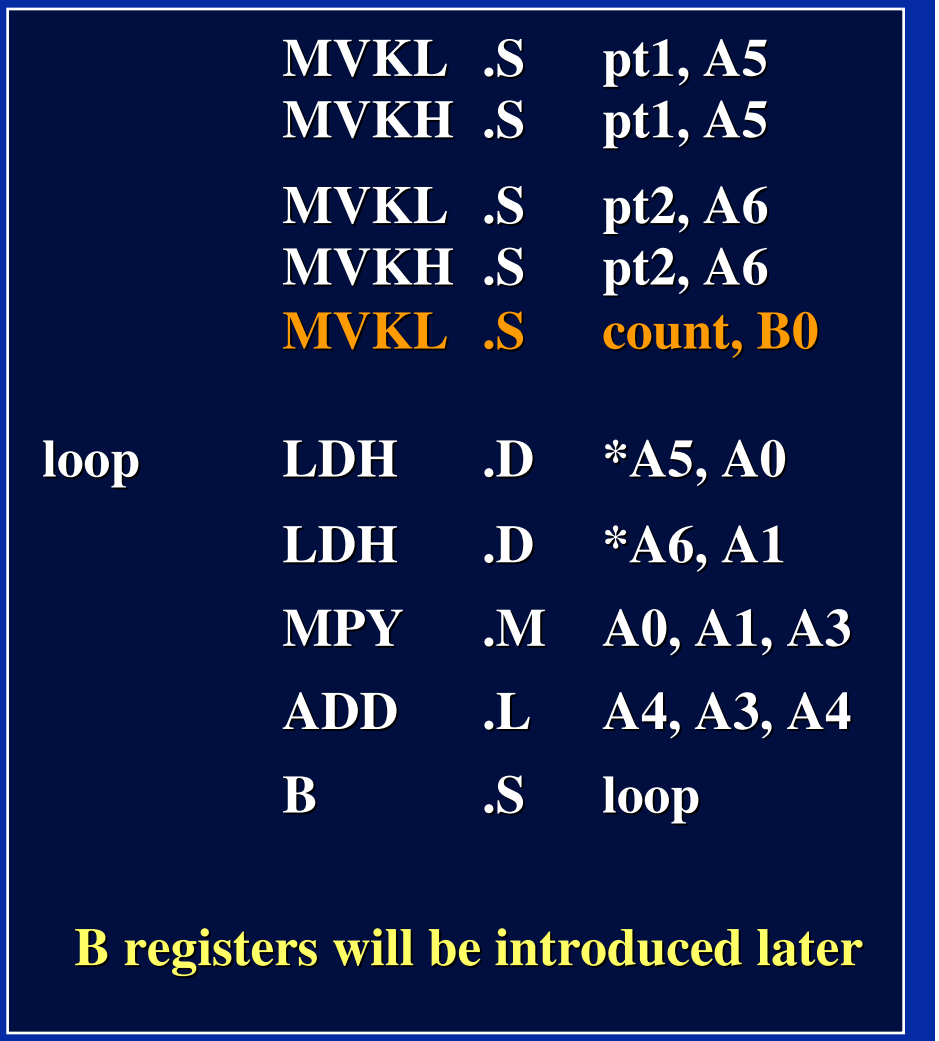



```
                MVKL  .S   pt1, A5
                MVKH  .S   pt1, A5

                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
```

Data Memory

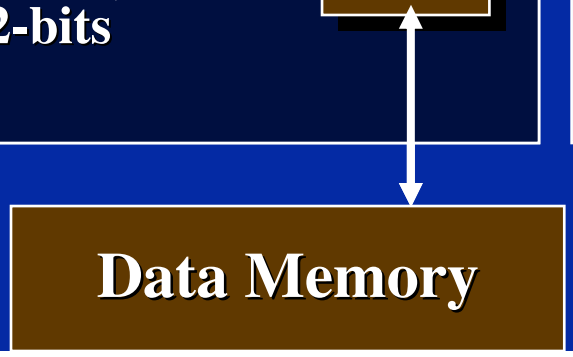
3. Create a loop counter.



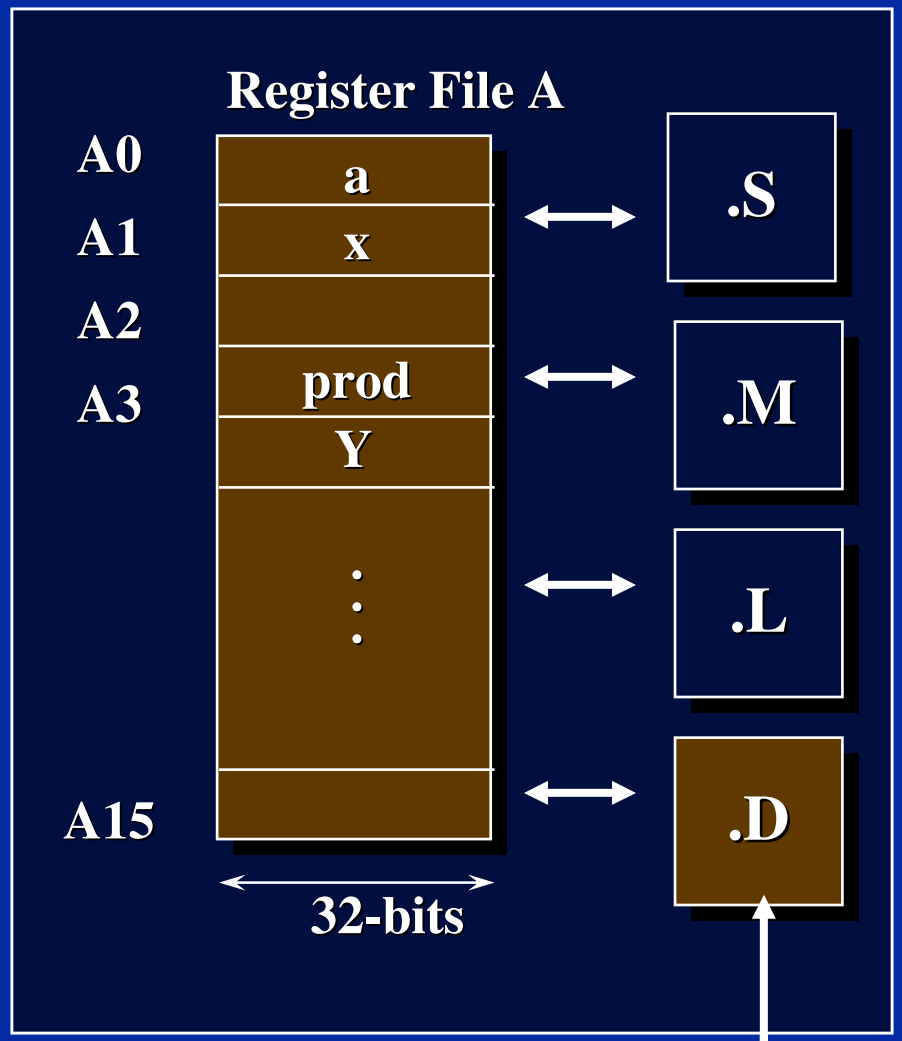


```
        MVKL  .S   pt1, A5
        MVKH  .S   pt1, A5

        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

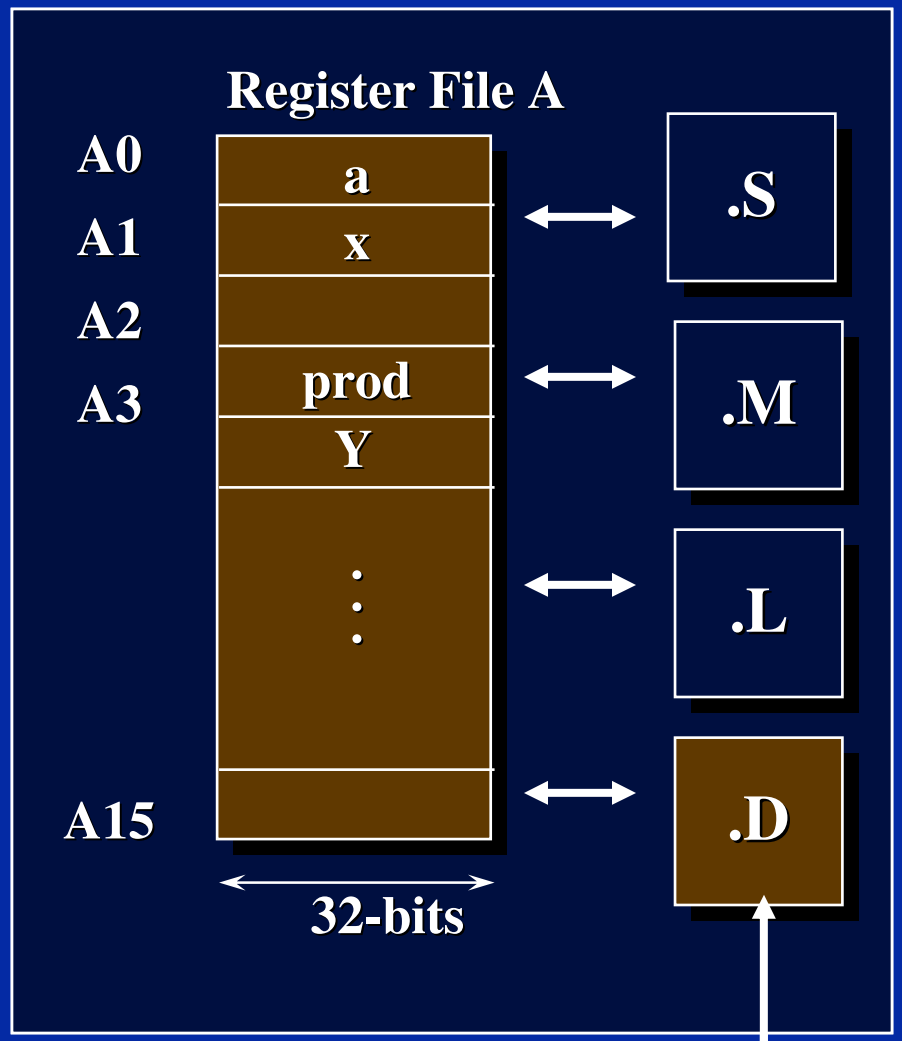
- The condition can be inverted by adding the exclamation symbol "!" as follows:

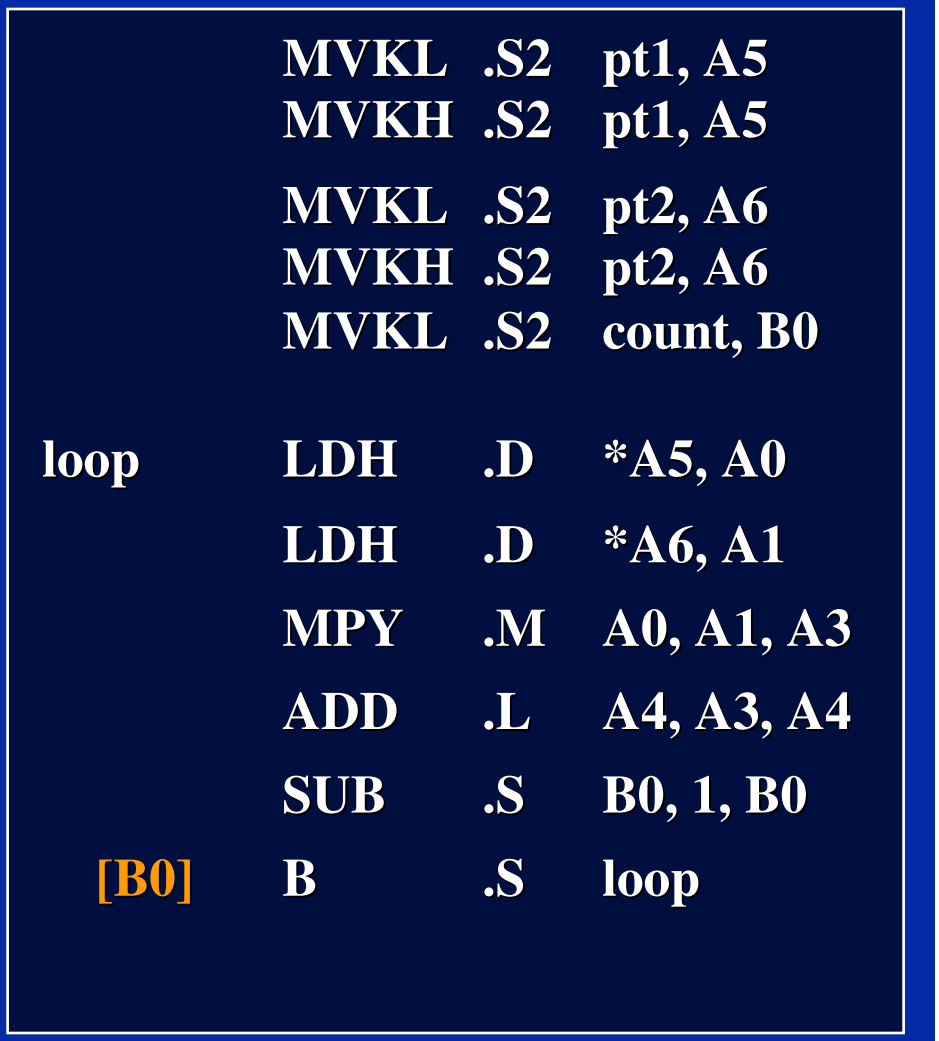
```
[!condition]    Instruction        Label
```

e.g.

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

5. Make the branch conditional

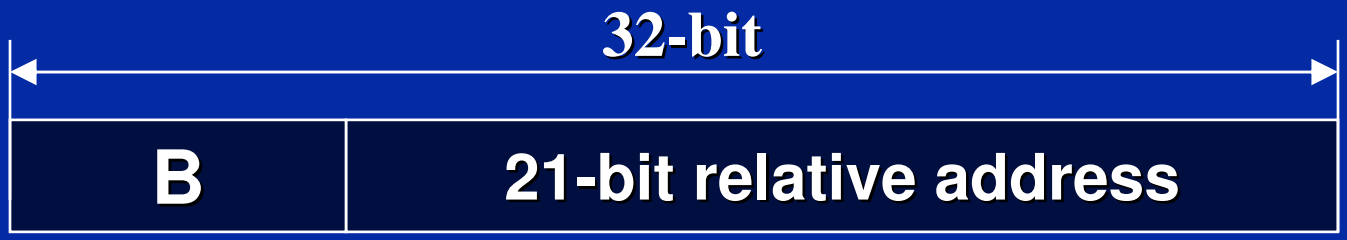




```
        MVKL  .S2   pt1, A5
        MVKH  .S2   pt1, A5

        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)

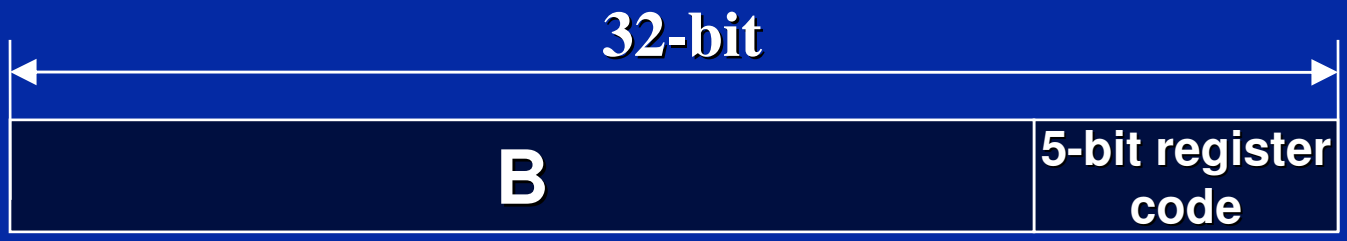
- With this processor all the instructions are encoded in a 32-bit.
- Therefore the label must have a dynamic range of less than 32-bit as the instruction B has to be coded.



- 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.

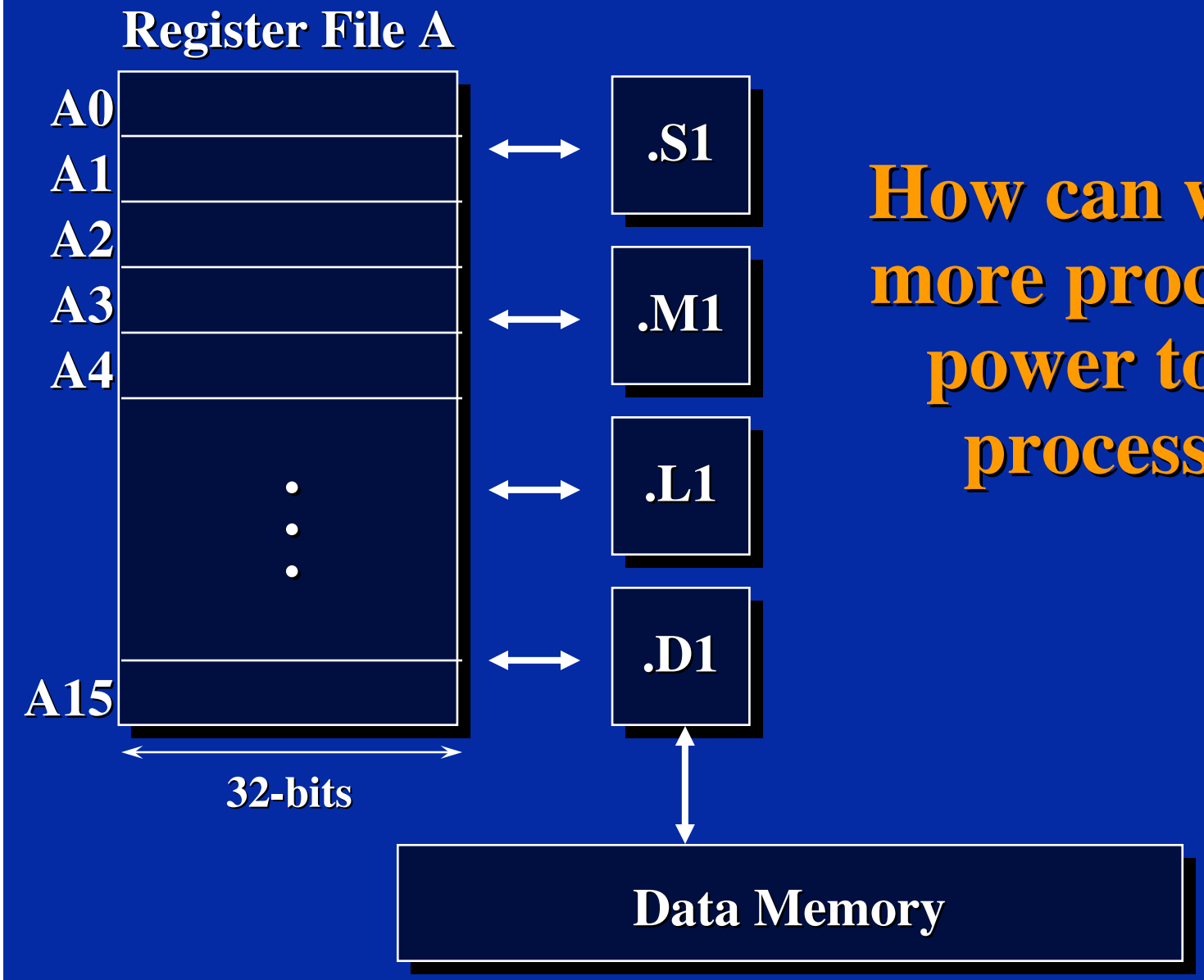
```
        MVKL   .S2   pt1, A5
        MVKH   .S2   pt1, A5

        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.

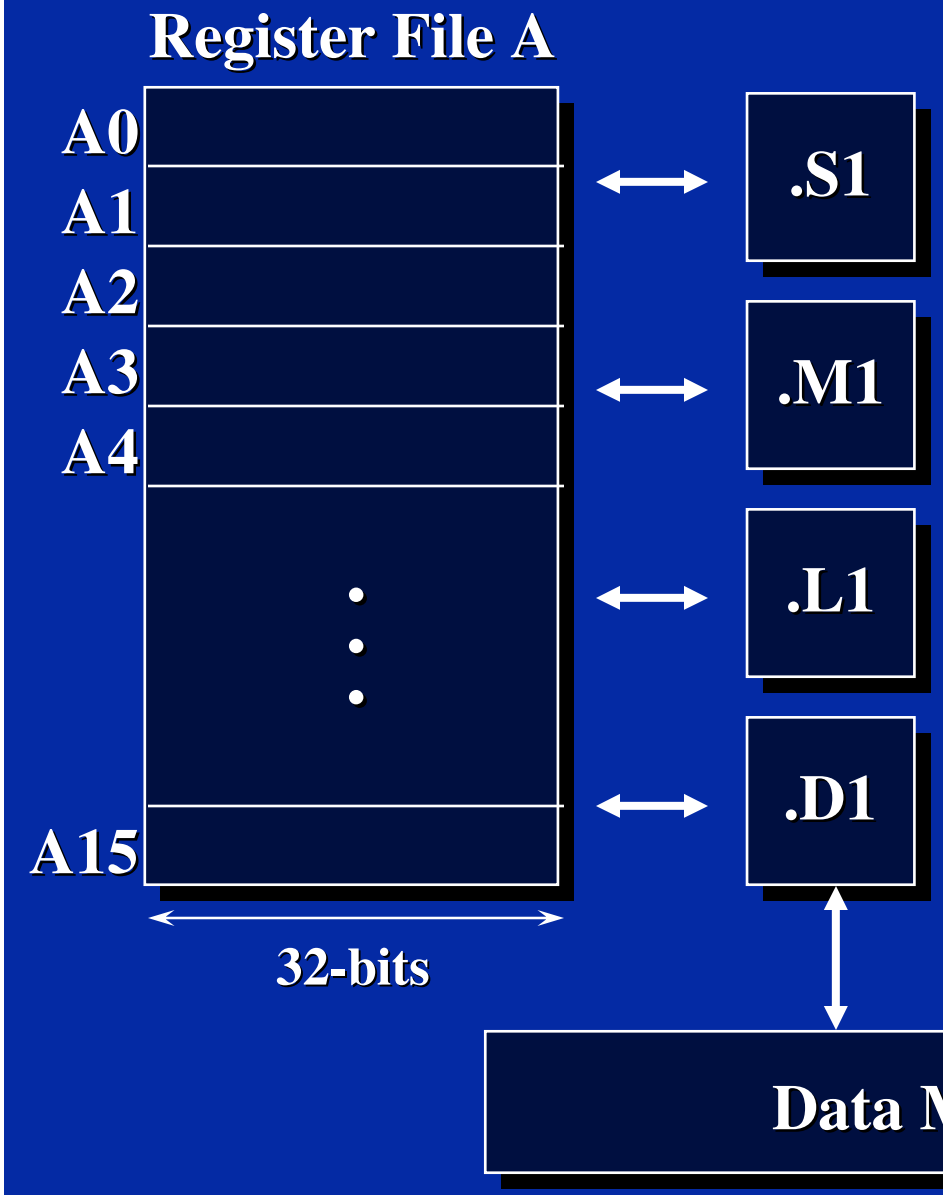
```
        MVKL   .S2   pt1, A5
        MVKH   .S2   pt1, A5

        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?

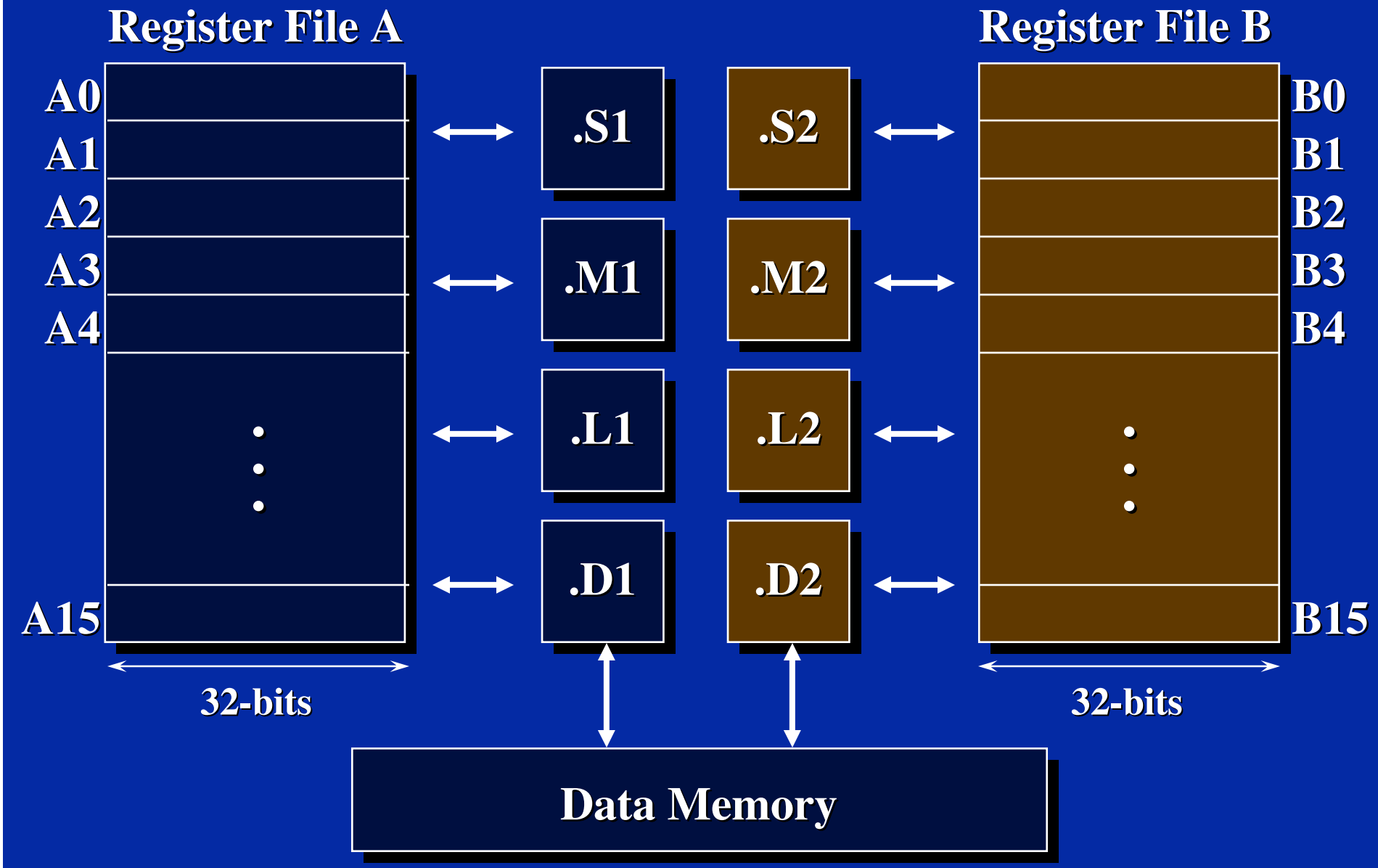
Increasing the processing power!



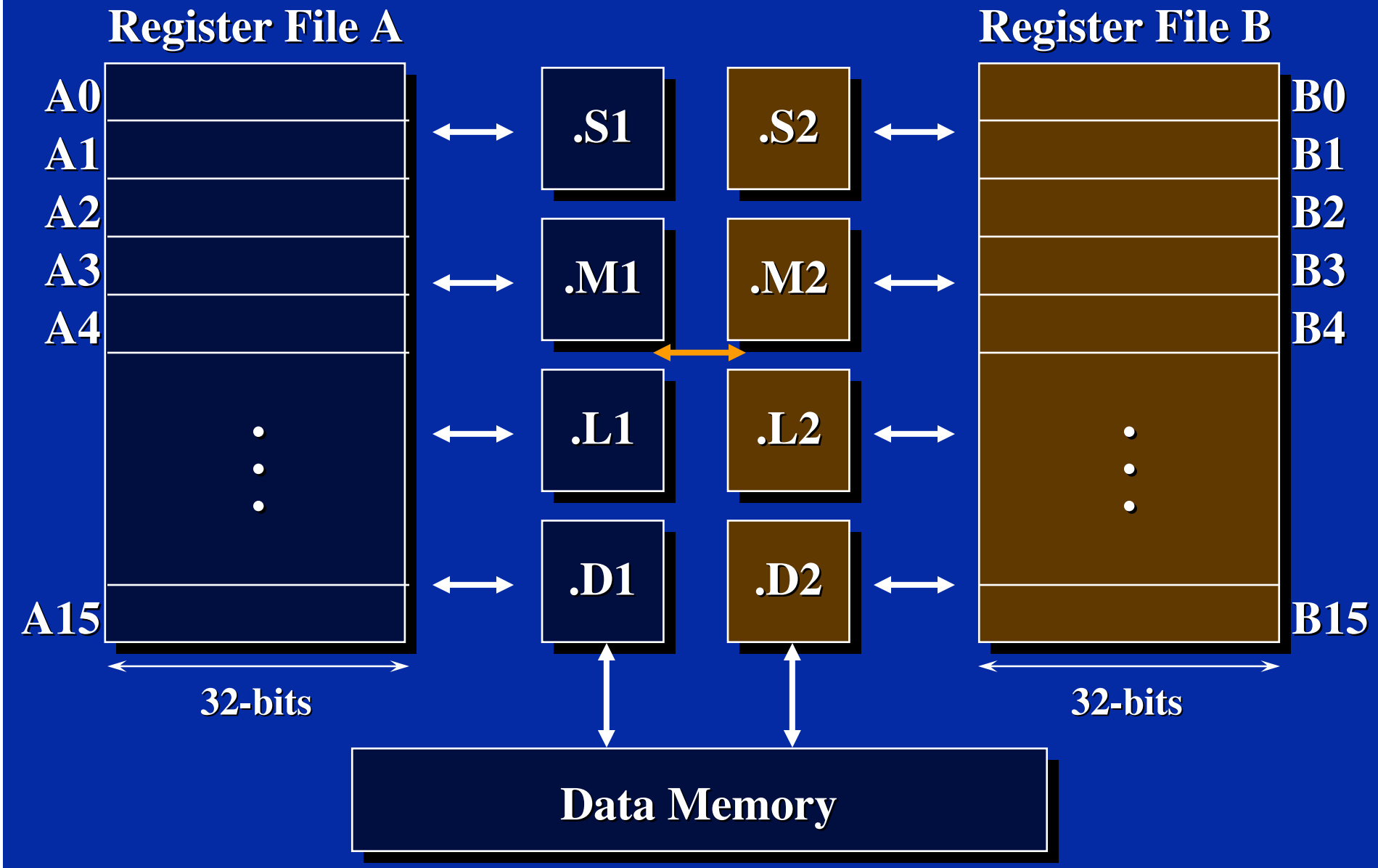
(1) Increase the clock frequency.

(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.

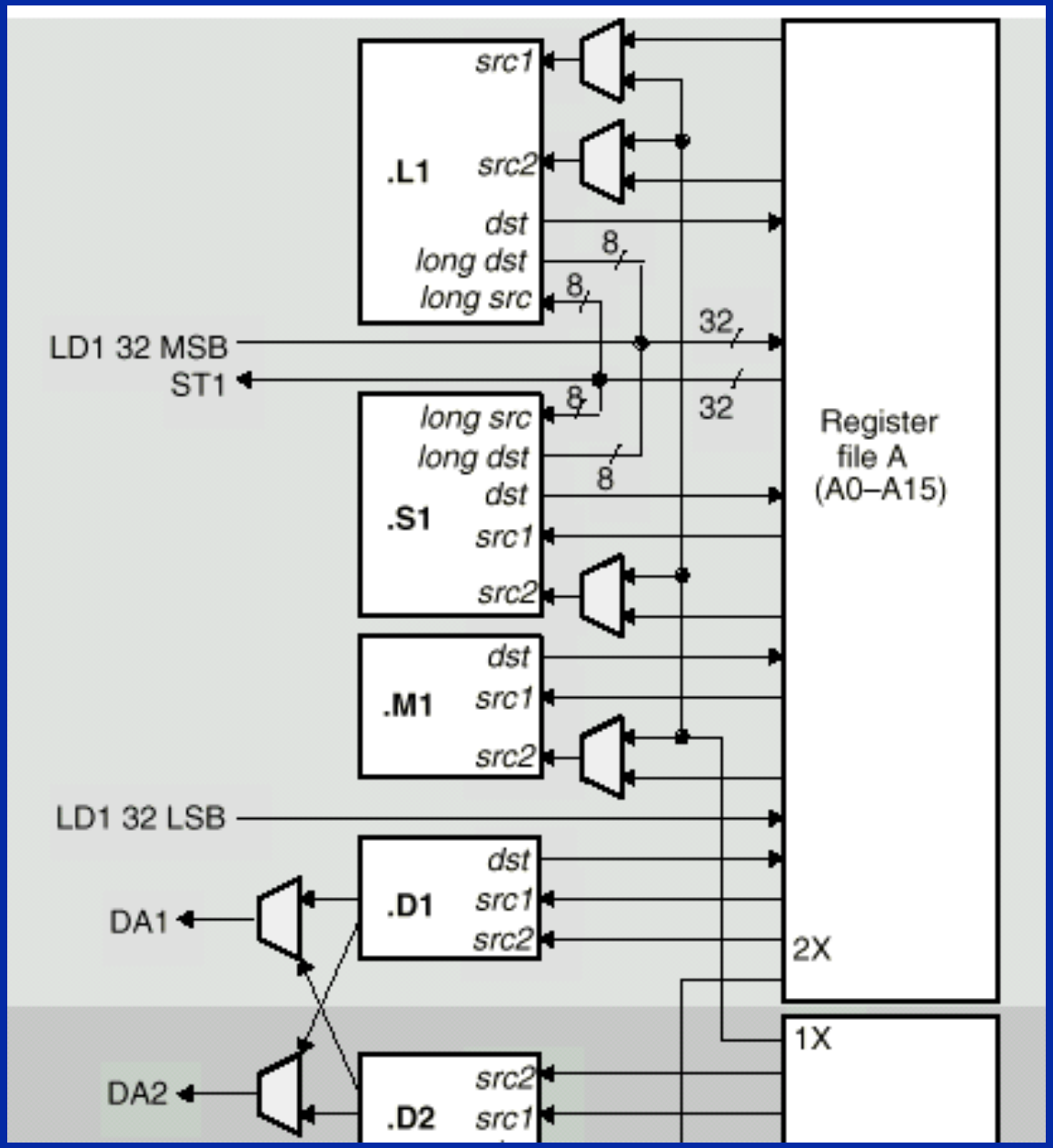
What is a cross path?

- 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.

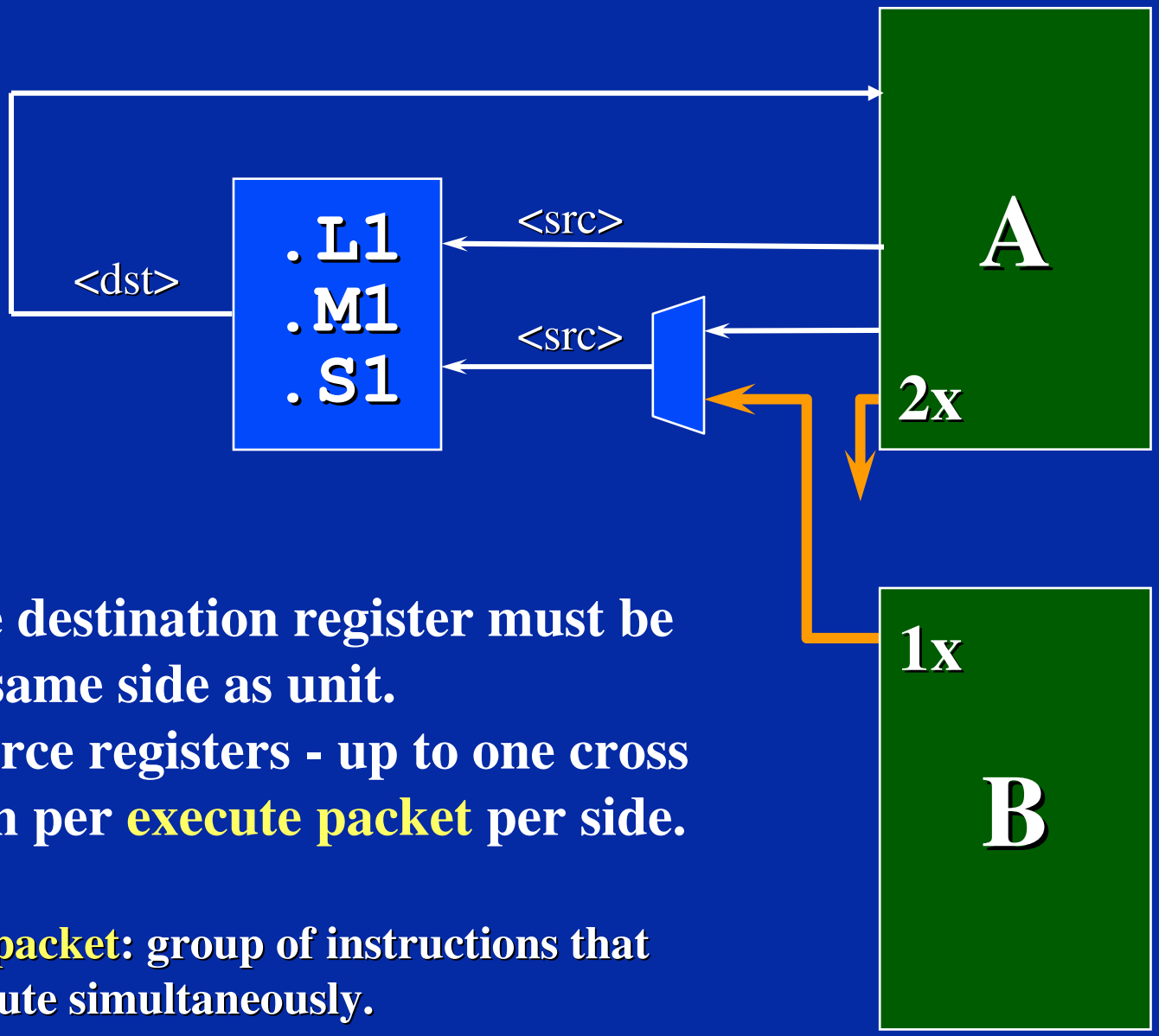
TMS320C67x Data-Path



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

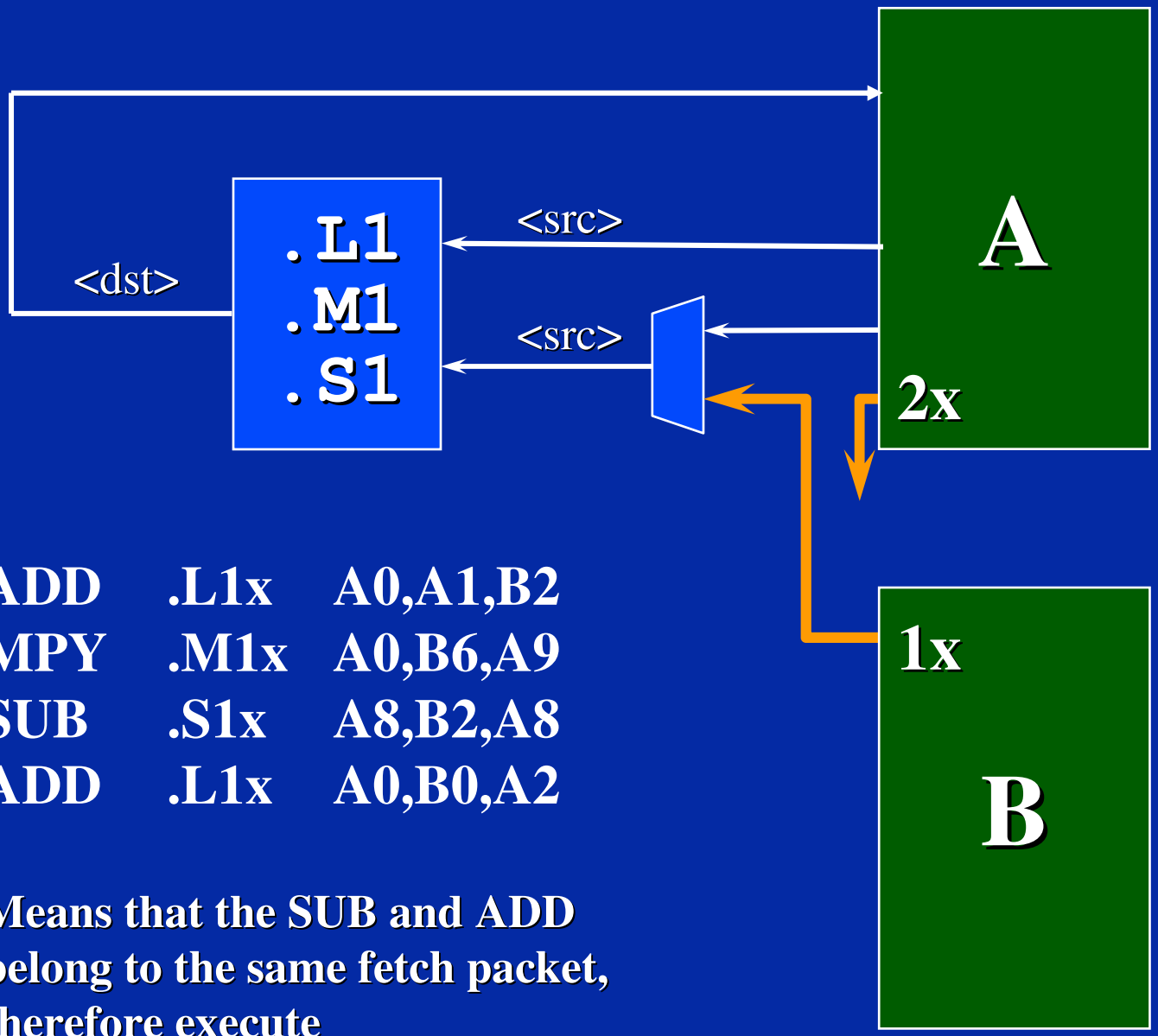


(1) The destination register must be on same side as unit.

(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

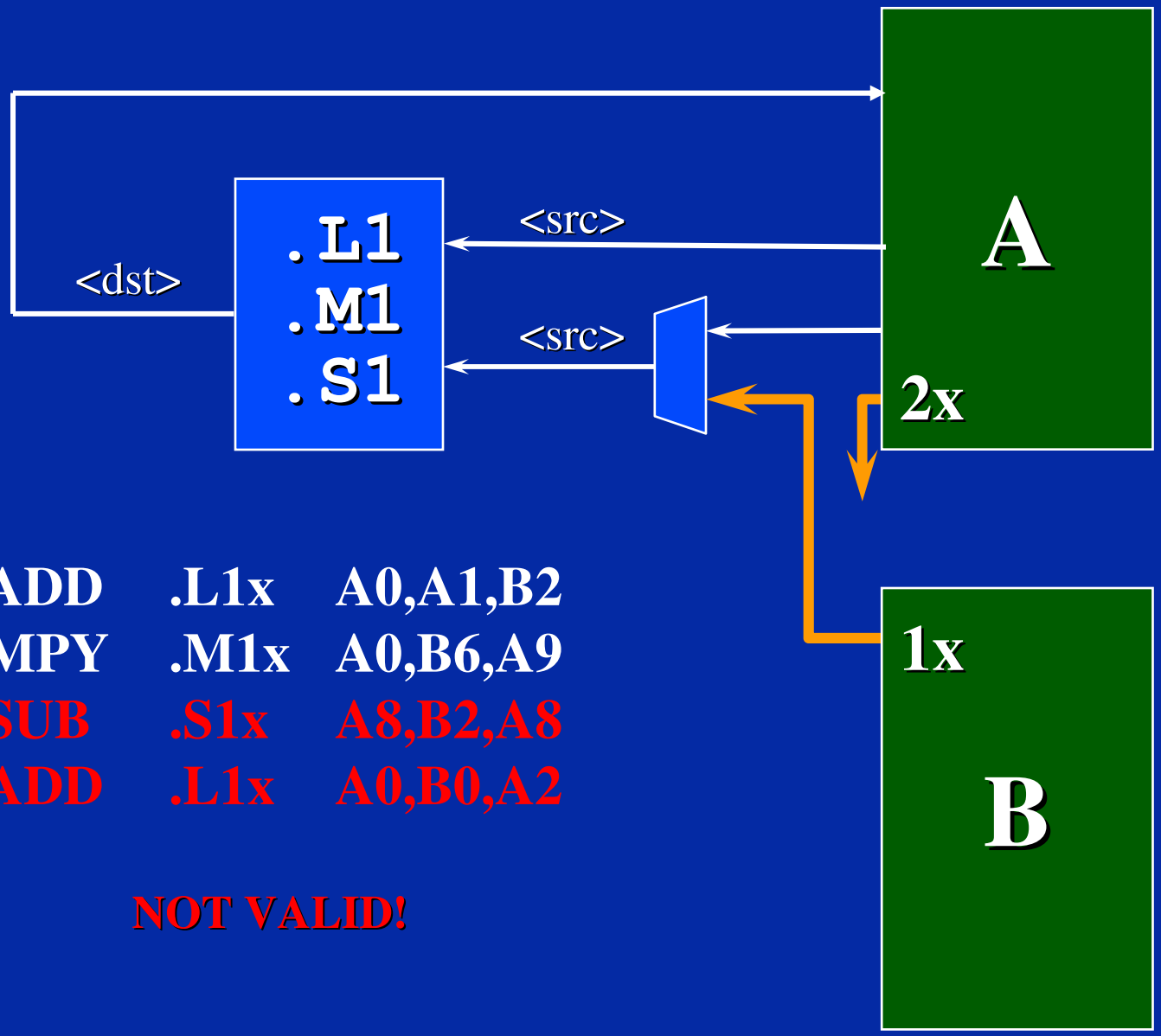


eg:

```
    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

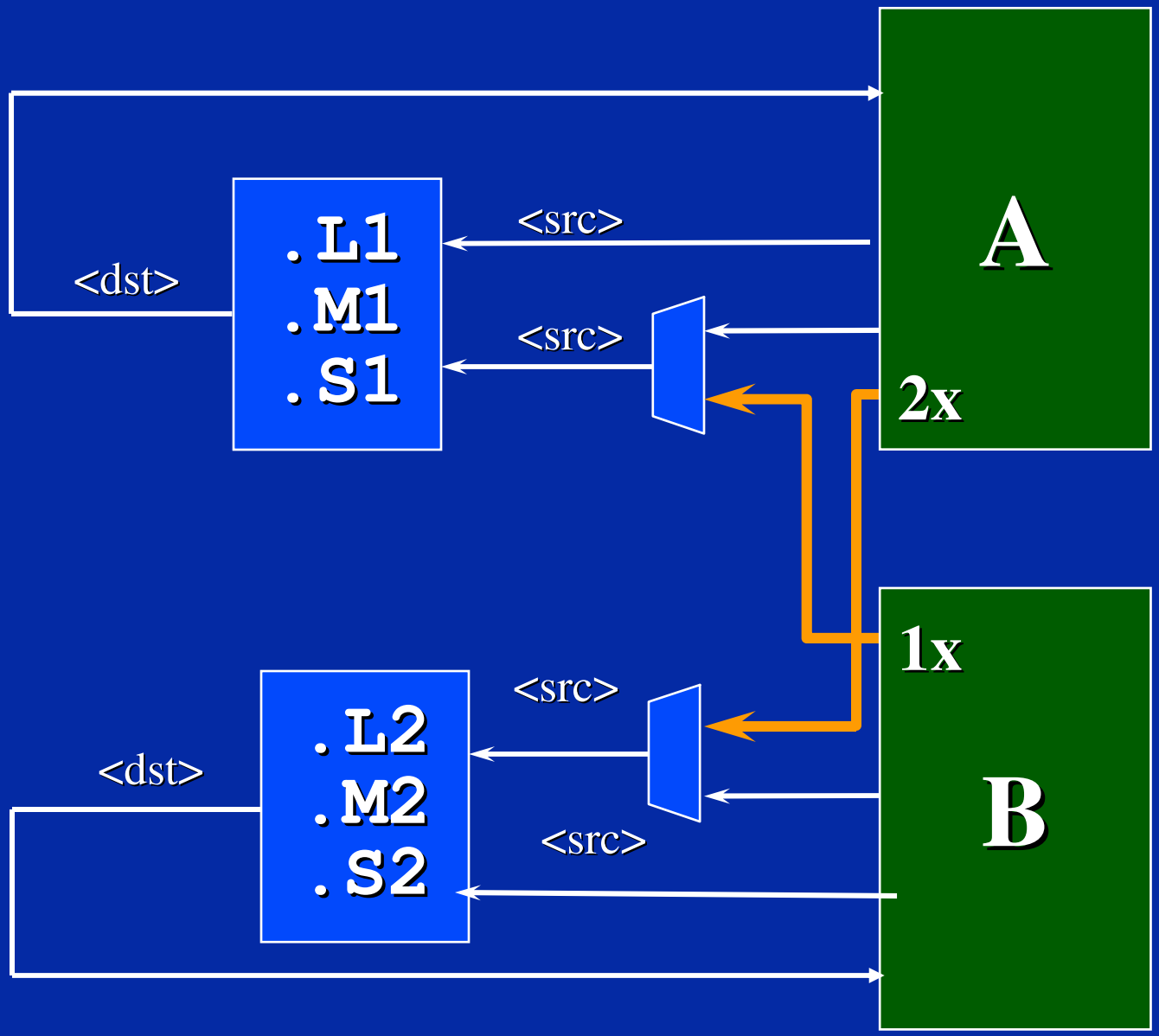


eg:

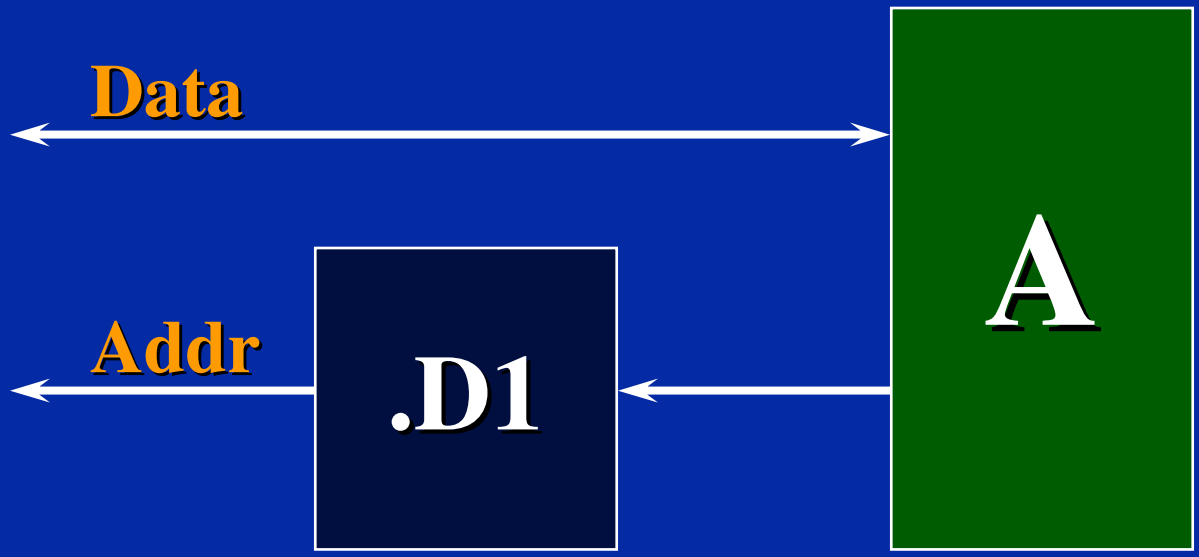
```
    ADD    .L1x   A0,A1,B2
    MPY    .M1x   A0,B6,A9
    SUB    .S1x   A8,B2,A8
||  ADD    .L1x   A0,B0,A2
```

NOT VALID!

Data Cross Paths for both sides



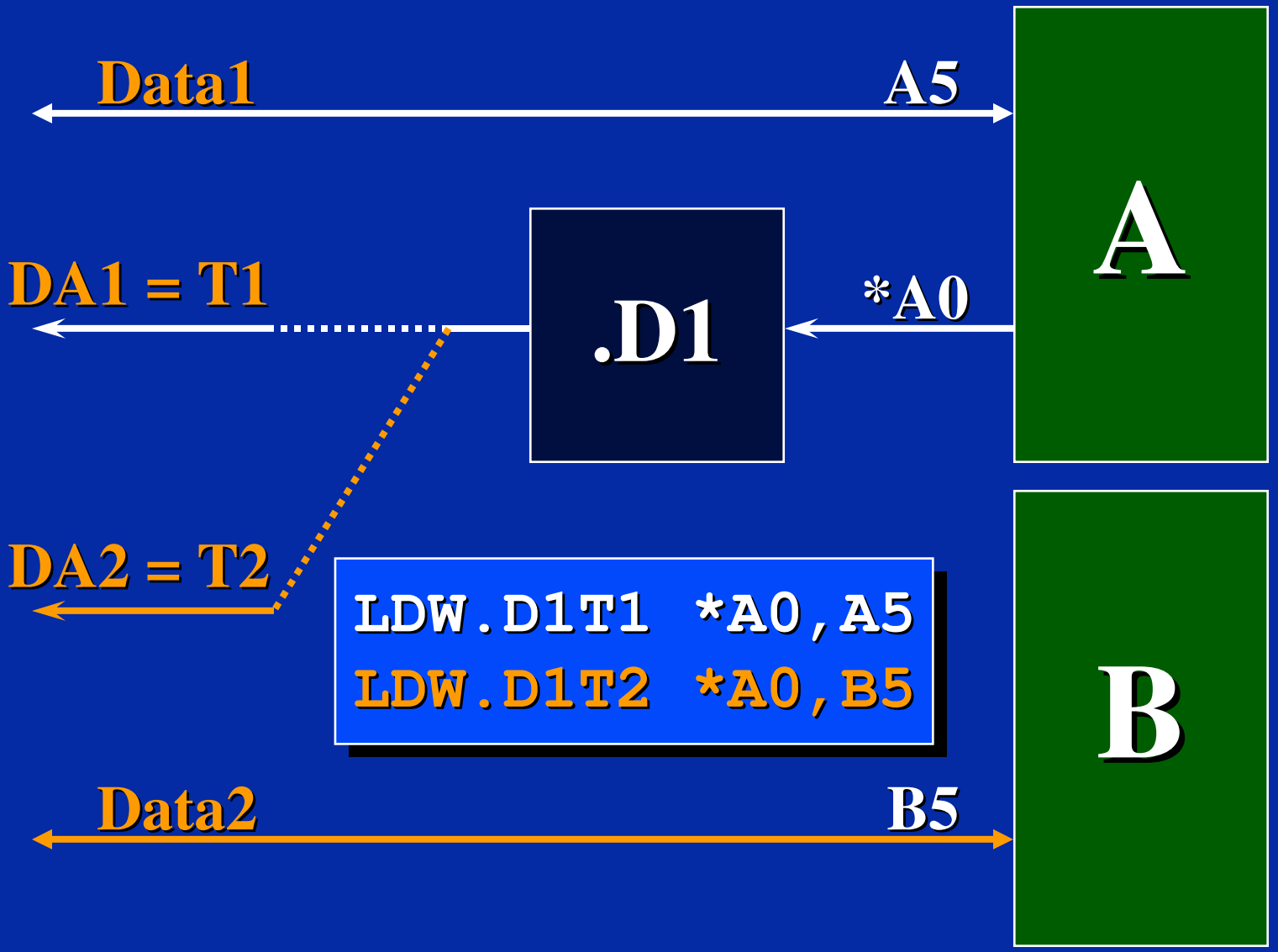
Address cross paths



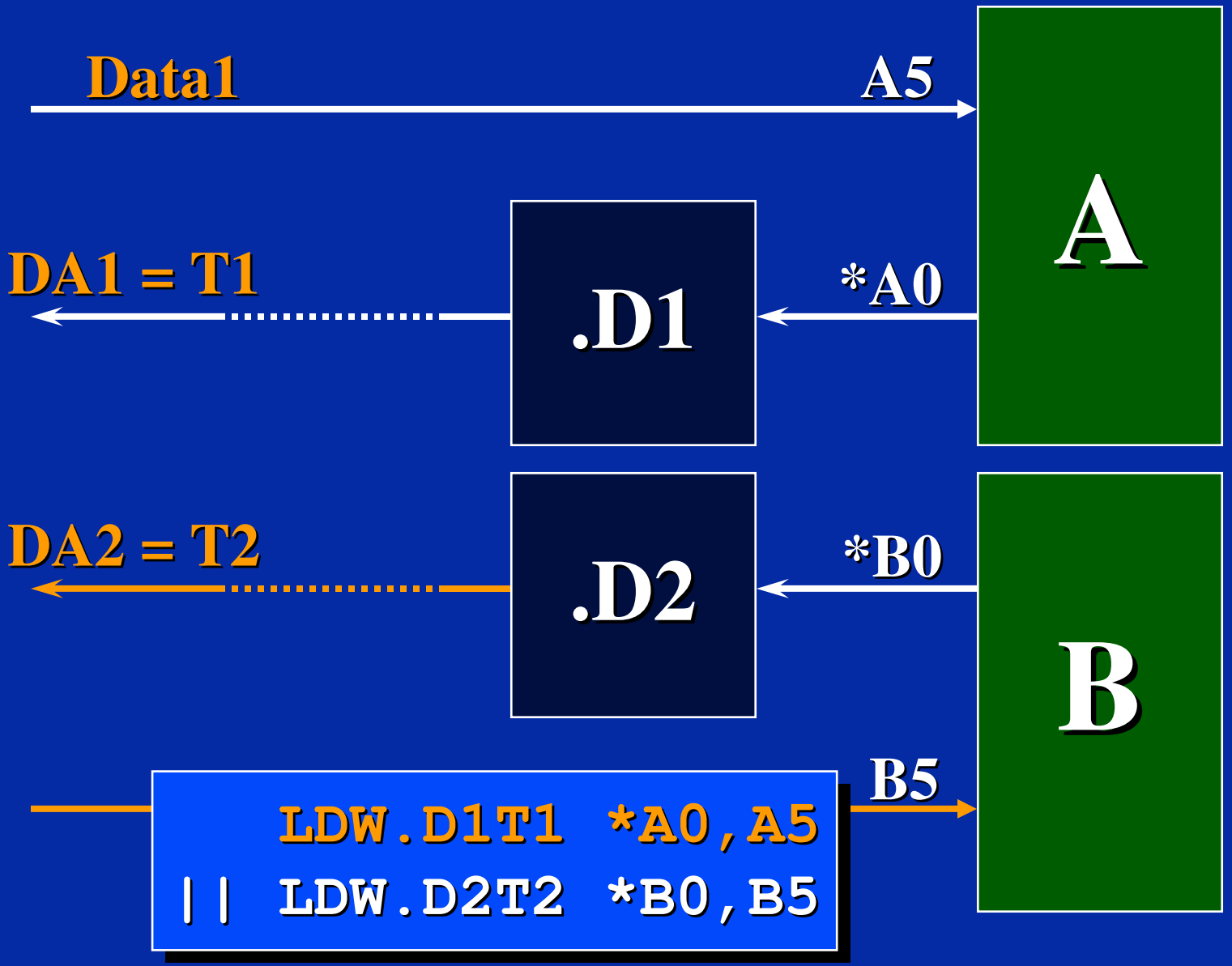
(1) The pointer must be on the same side of the unit.

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

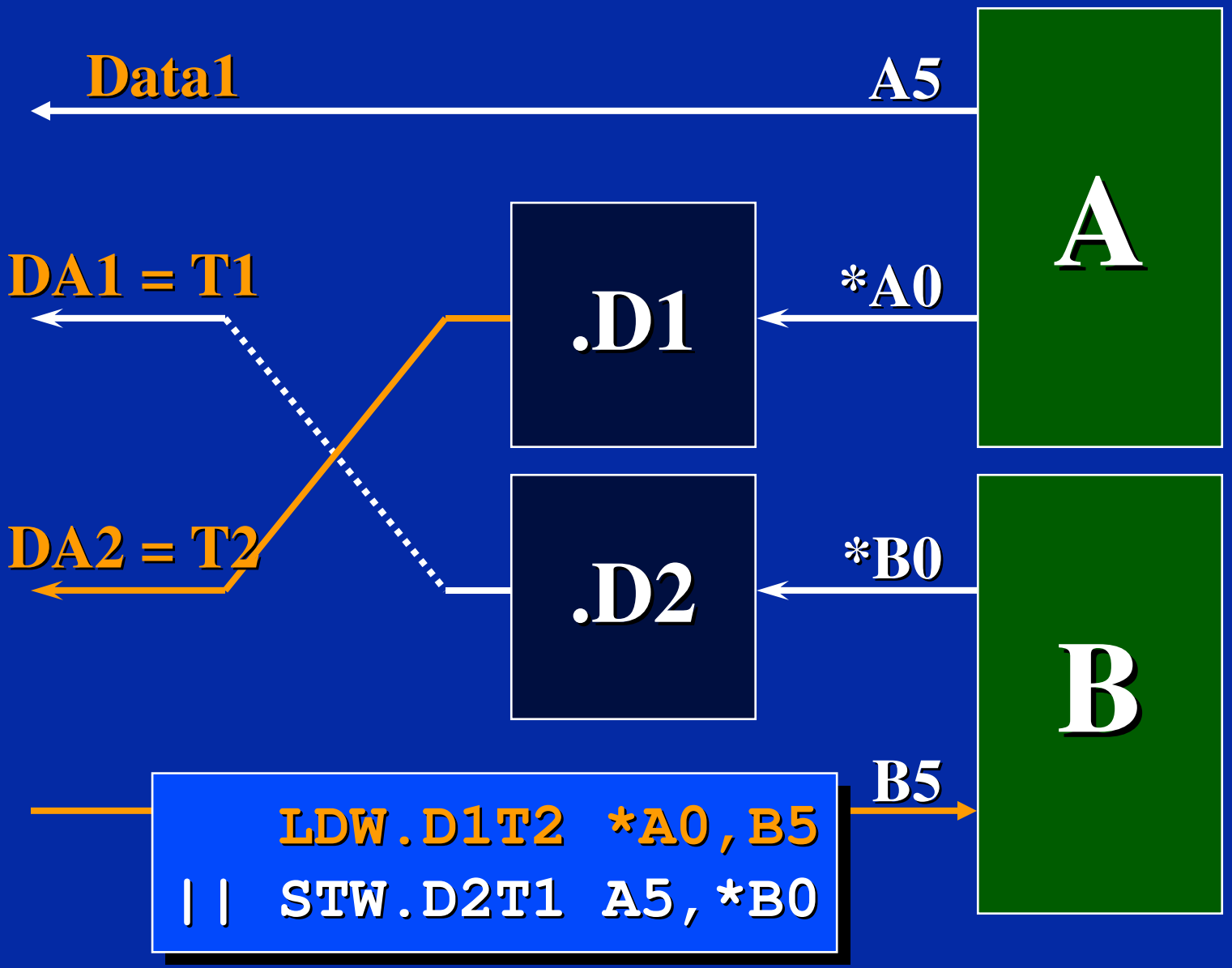
Load or store to either side



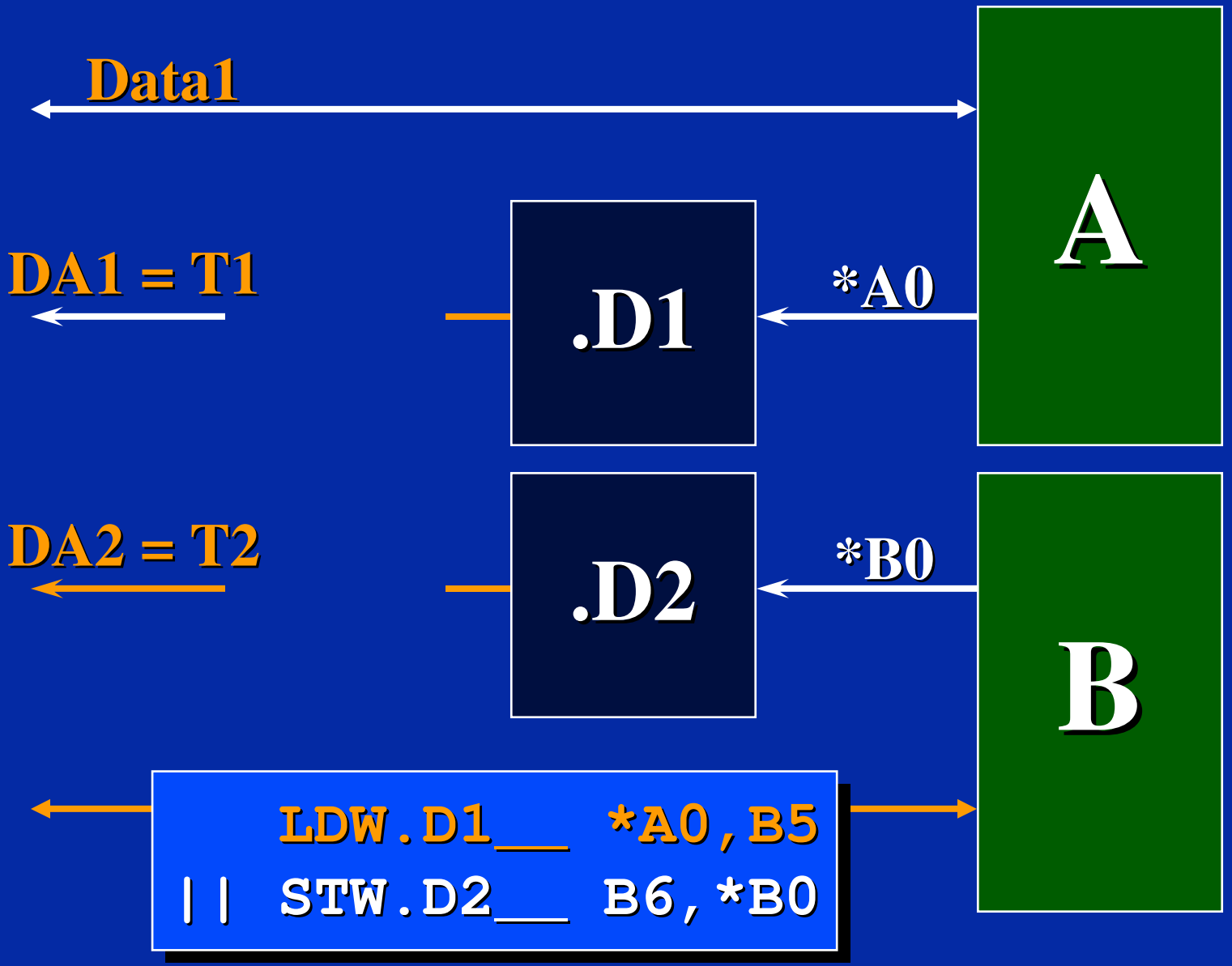
Standard Parallel Loads



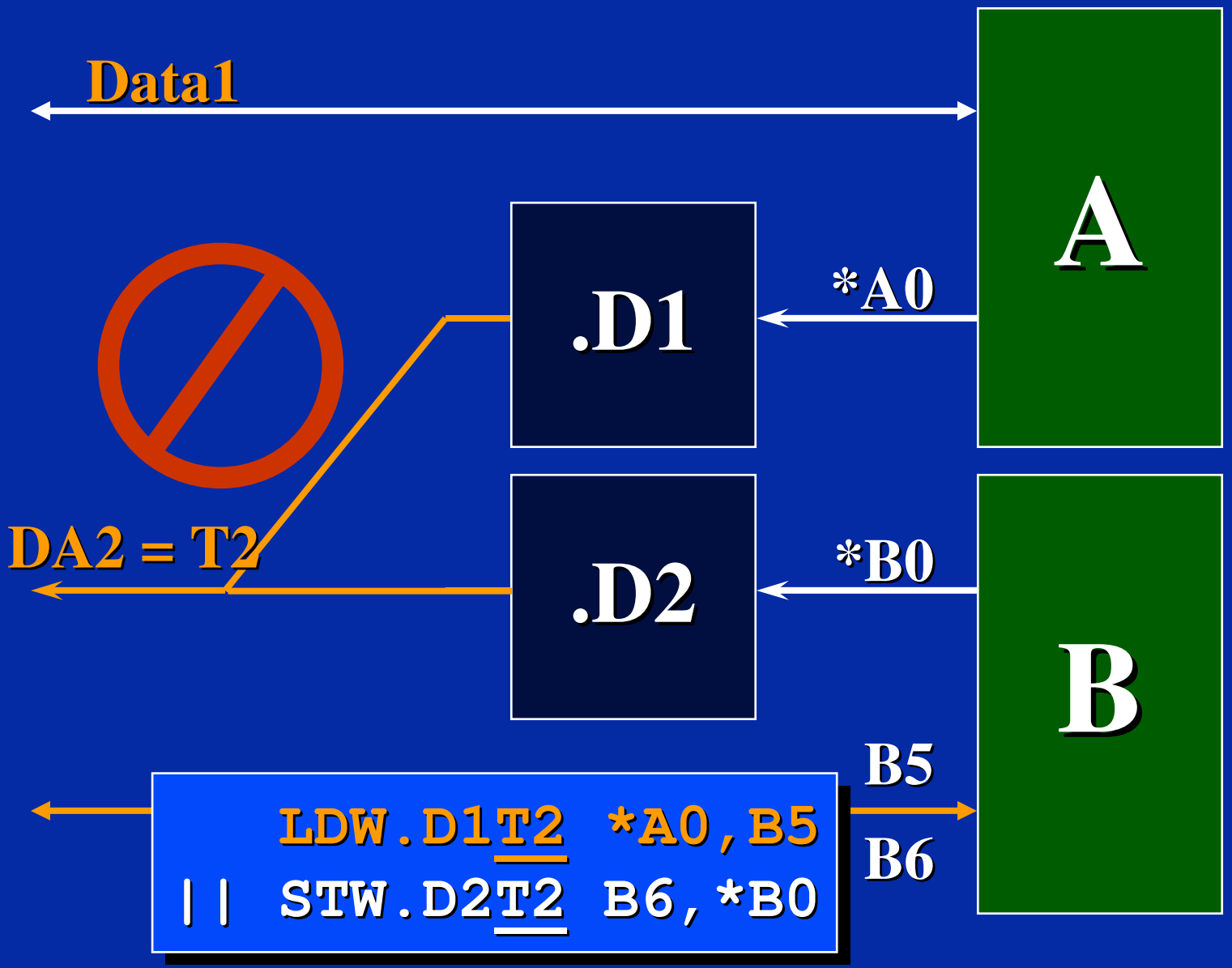
Parallel Load/Store using address cross paths



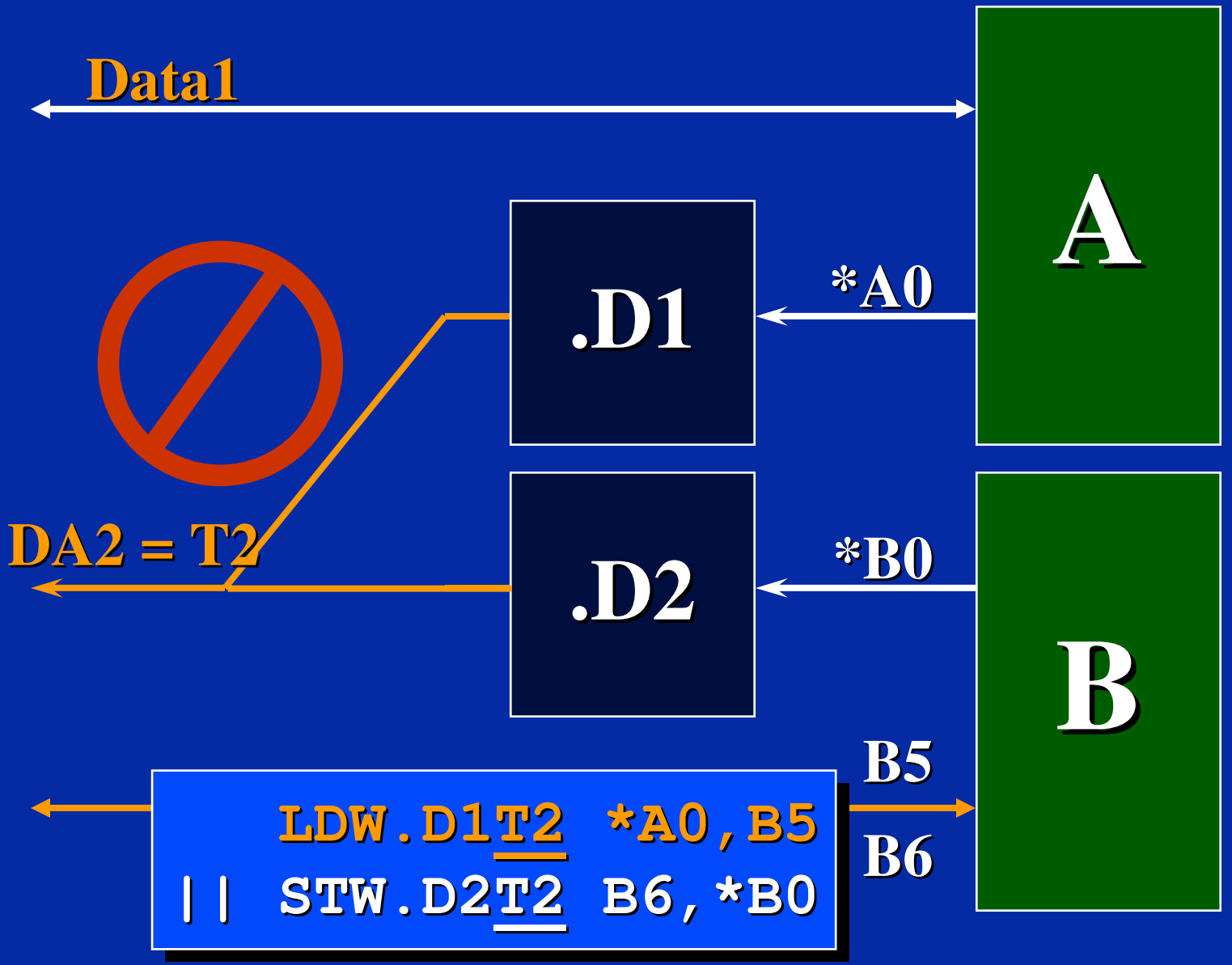
Fill the blanks ... Does this work?



Not Allowed!



Not Allowed!
Parallel accesses: both cross or neither cross

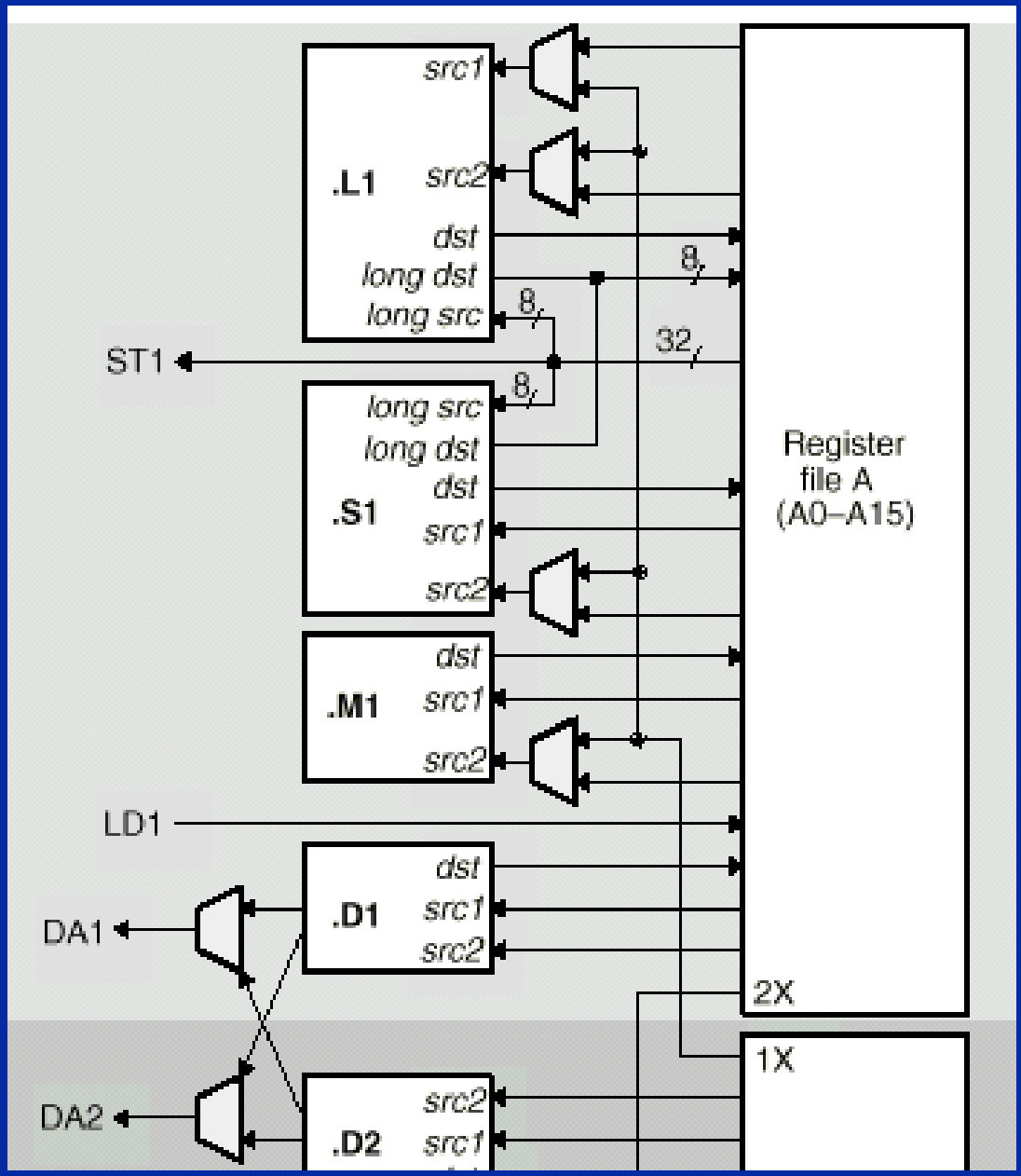


Conditions Don't Use Cross Paths

- 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
```

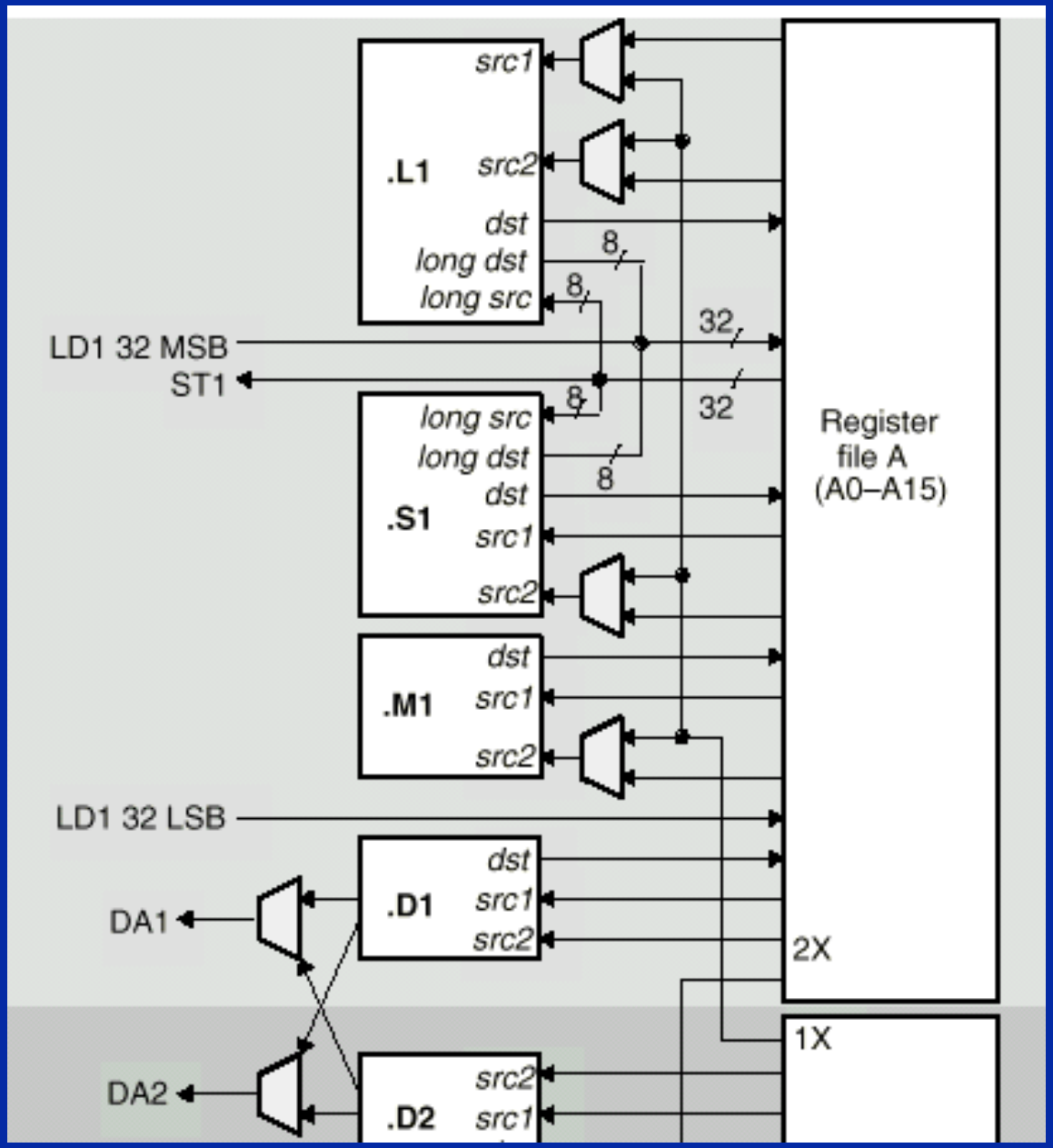
‘C62x Data-Path Summary



CPU
Ref Guide

Full CPU Datapath
(Pg 2-2)

‘C67x Data-Path Summary



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.

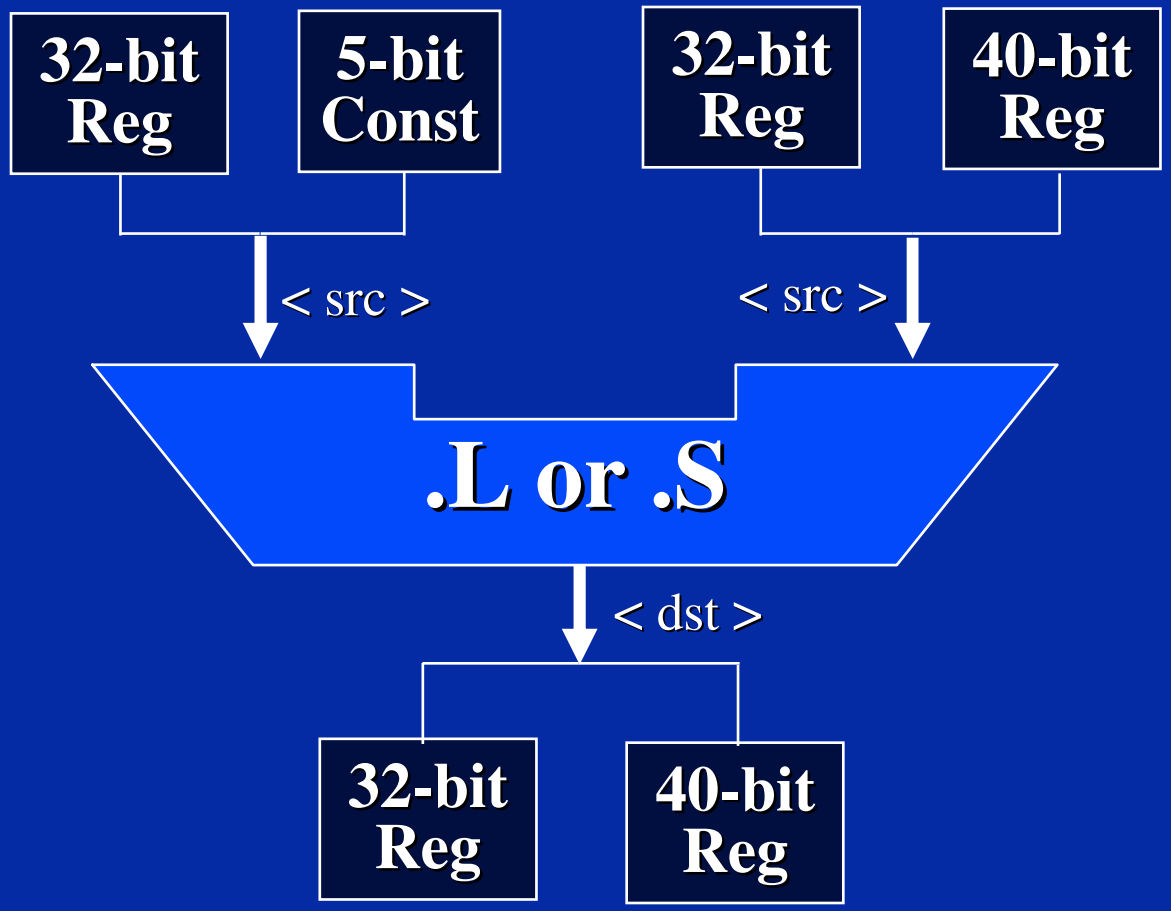
Operands - 32/40-bit Register, 5-bit Constant

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

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

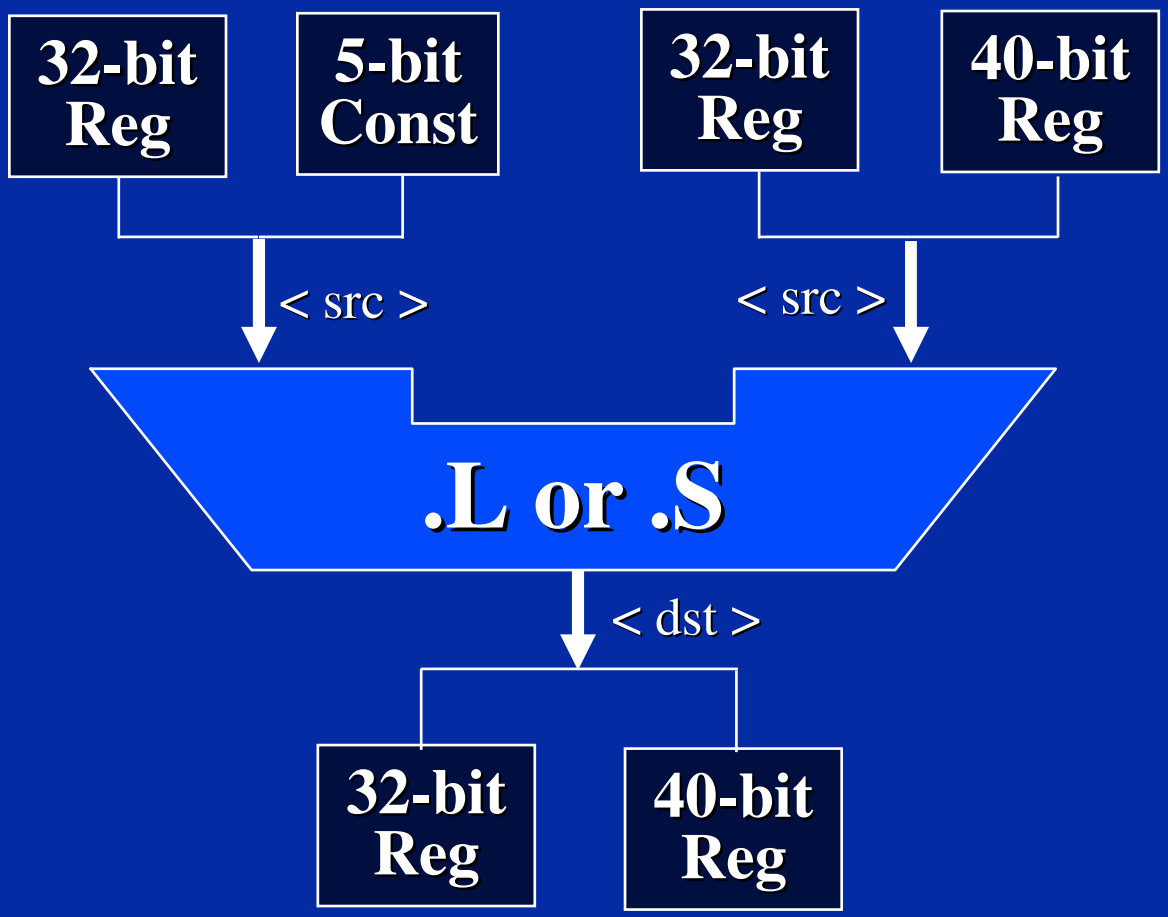
Operands - 32/40-bit Register, 5-bit Constant

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



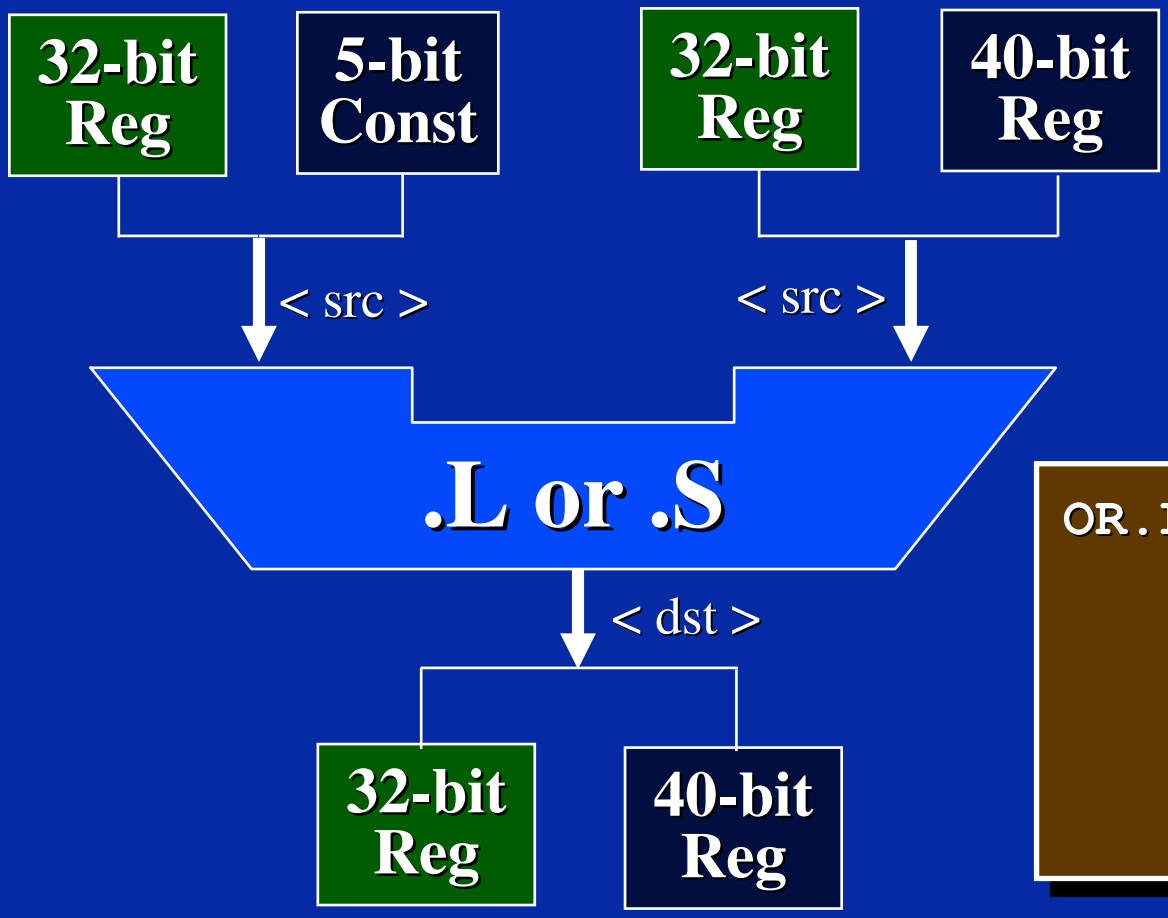
Operands - 32/40-bit Register, 5-bit Constant

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



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

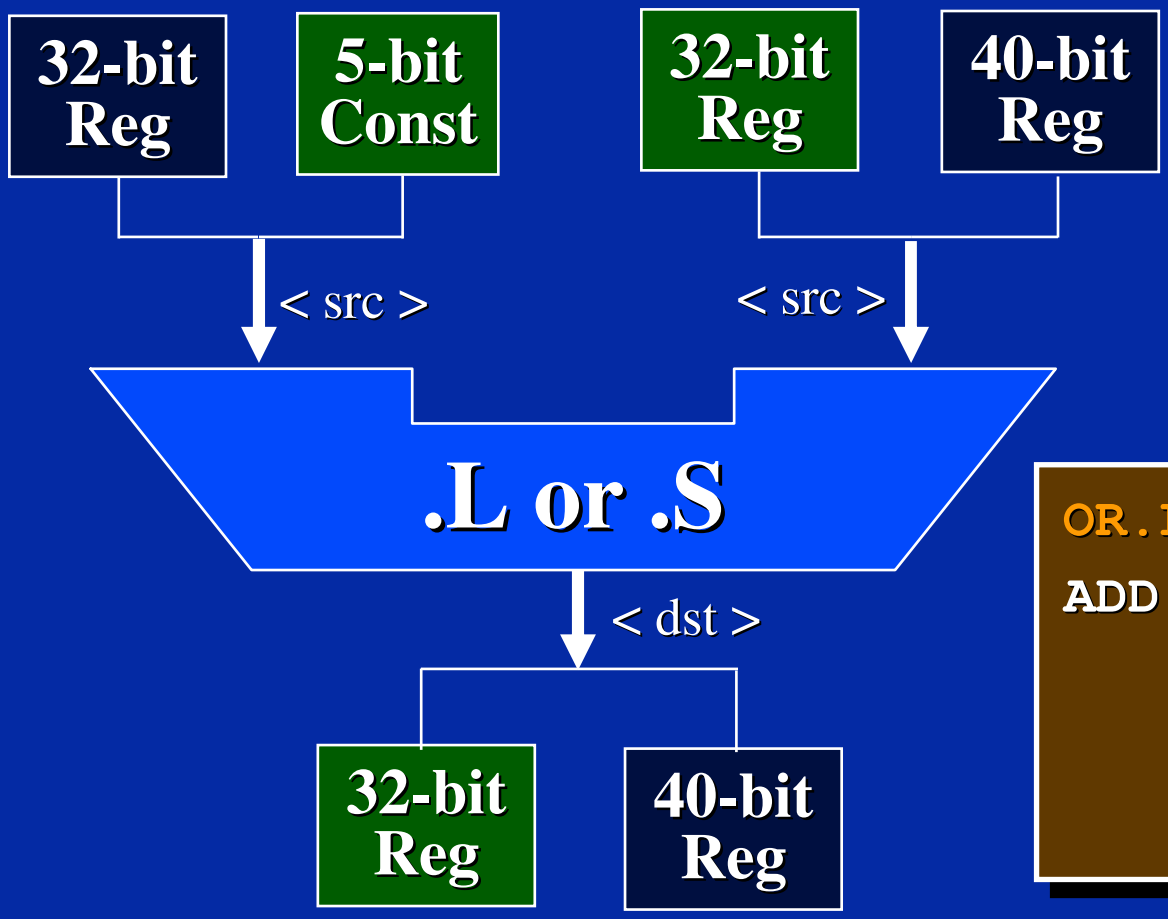
```
instr   .unit   <src>, <src>, <dst>
```




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

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

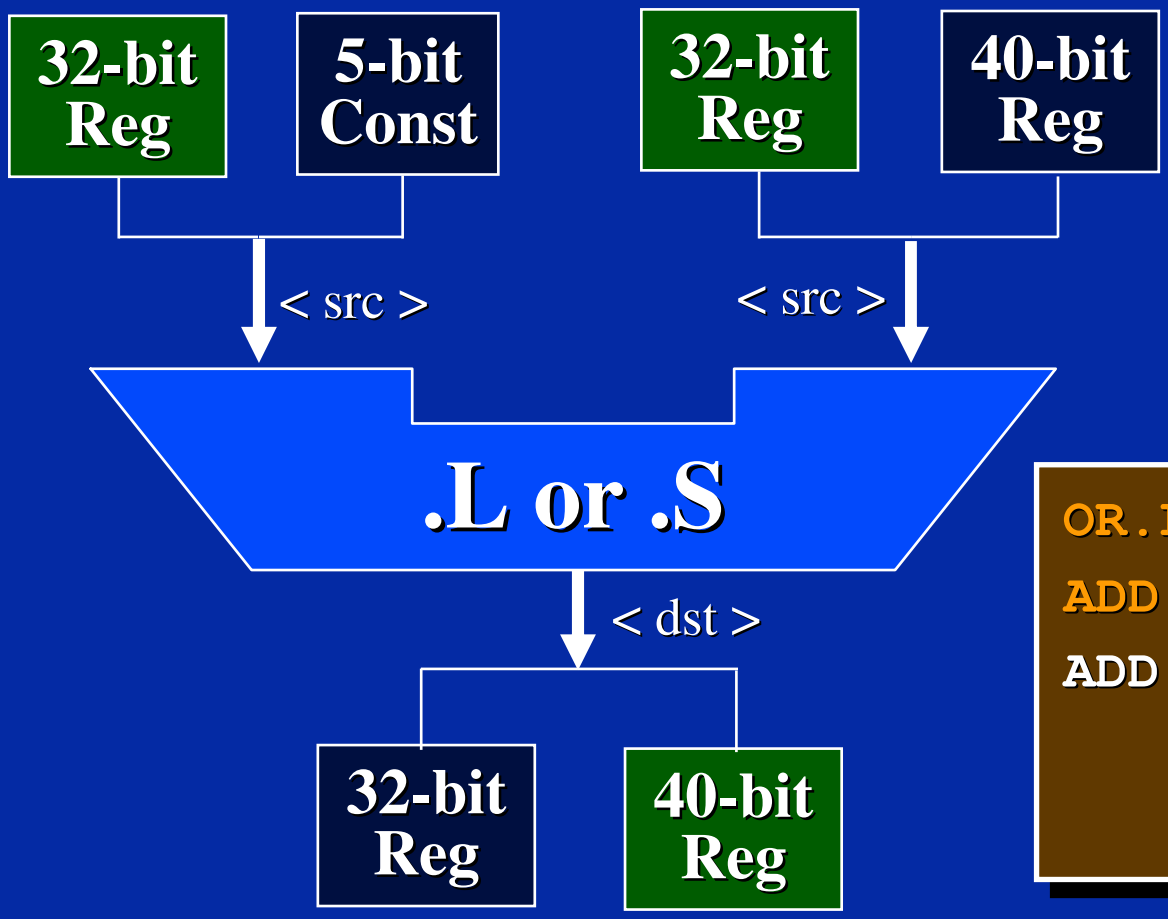
```
instr   .unit   <src>, <src>, <dst>
```




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

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

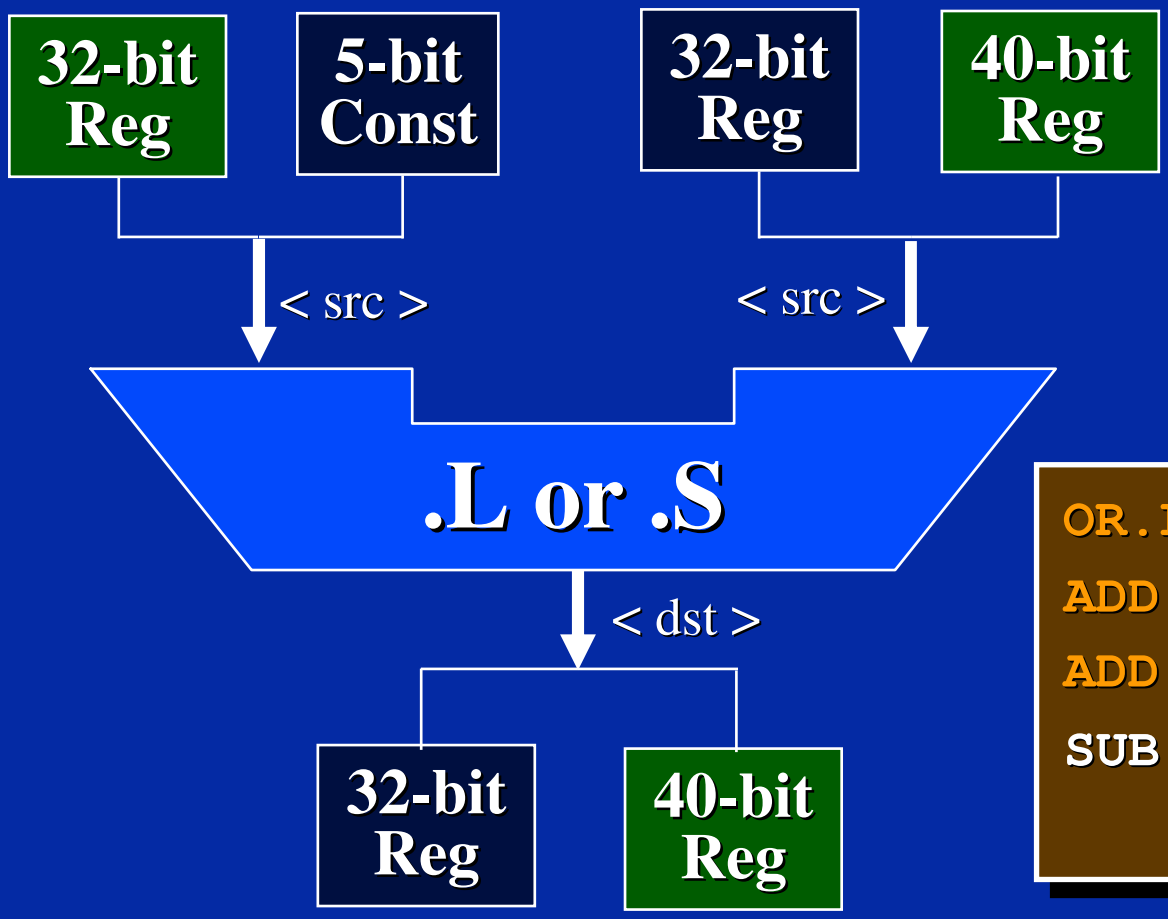
```
instr   .unit   <src>, <src>, <dst>
```




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

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

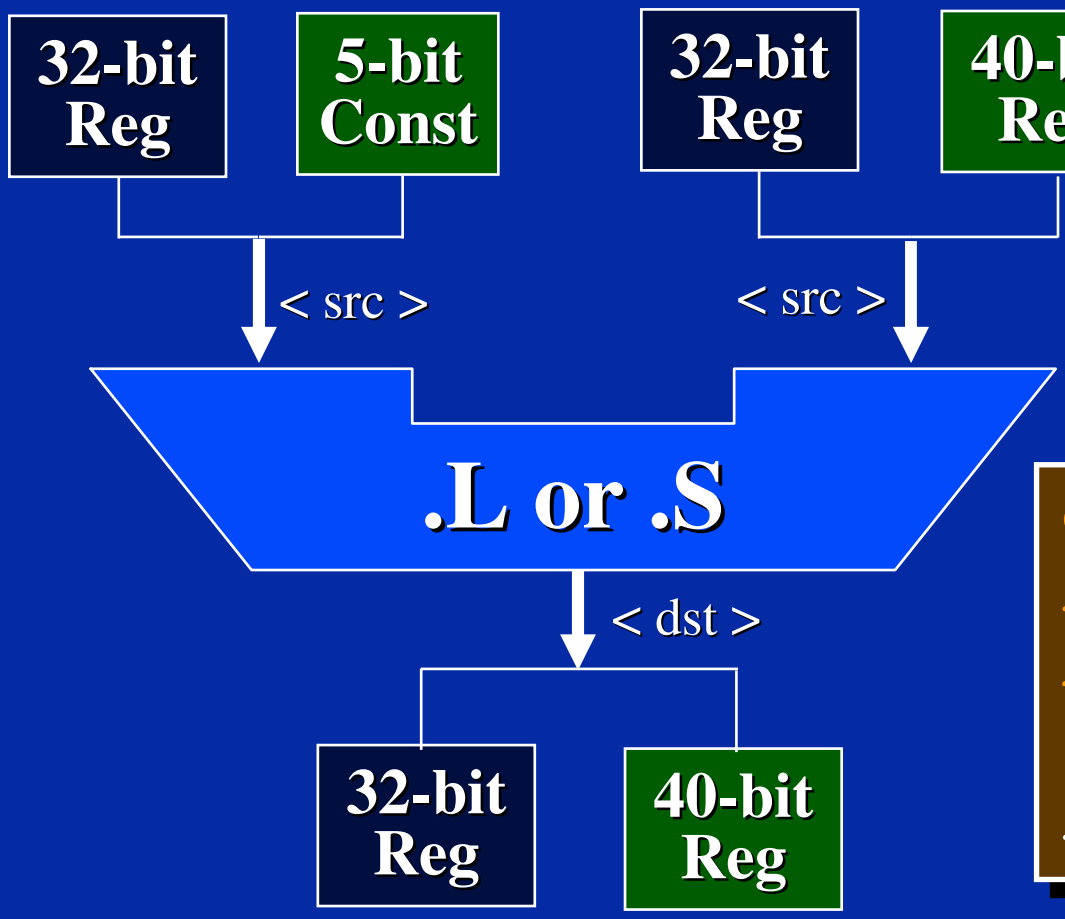
```
instr   .unit   <src>, <src>, <dst>
```




```
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
```

TMS320C6211/6711 Instruction Set

'C6211 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.

'C6211 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.

‘C6711 Additional Instructions (by unit)

.S Unit	
ABSSP	CMPLTDP
ABSDP	RCPSP
CMPGTSP	RCPDP
CMPEQSP	RSQRSP
CMPLTSP	RSQRDP
CMPGTDP	SPDP
CMPEQDP	

.L Unit	
ADDDP	INTSP
ADDSP	INTSPU
DPINT	SPINT
DPSP	SPTRUNC
INTDP	SUBSP
INTDPU	SUBDP

.M Unit	
MPYSP	MPYI
MPYDP	MPYID

.D Unit	
ADDAD	LDDW

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

TMS320C6211/6711 Memory Map

‘C6211 Memory Map

External Memory

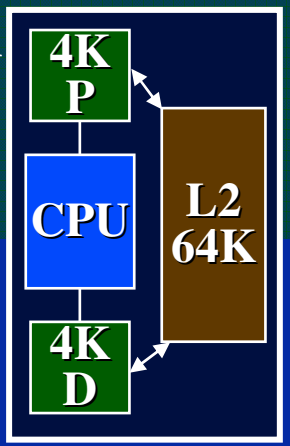
- Async (SRAM, ROM, etc.)
- Sync (SBSRAM, SDRAM)

Internal Memory

- Unified (data or prog)
- 4 blocks - each can be RAM or cache

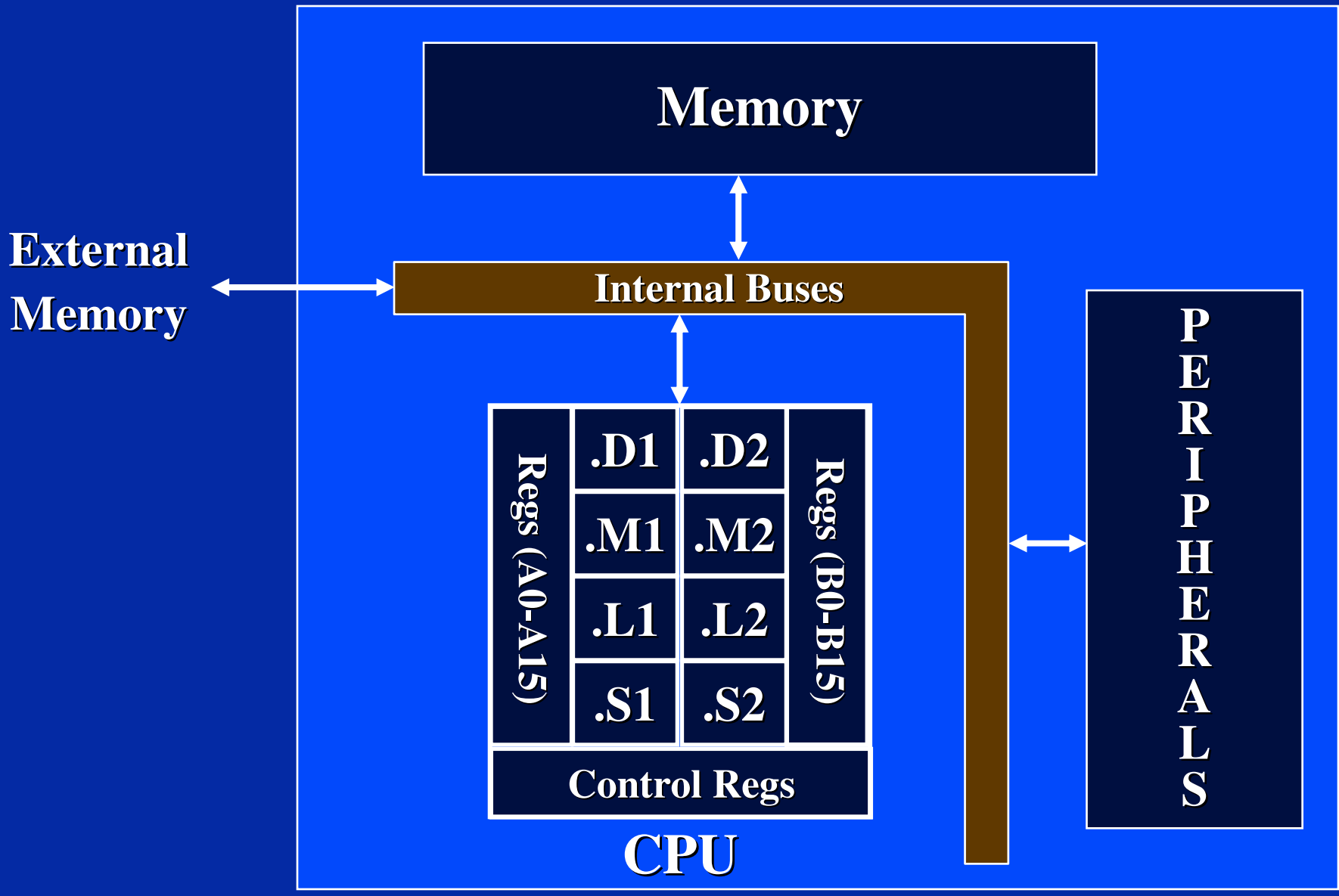
Level 1 Cache

- 4KB Program
- 4KB Data
- Not in map

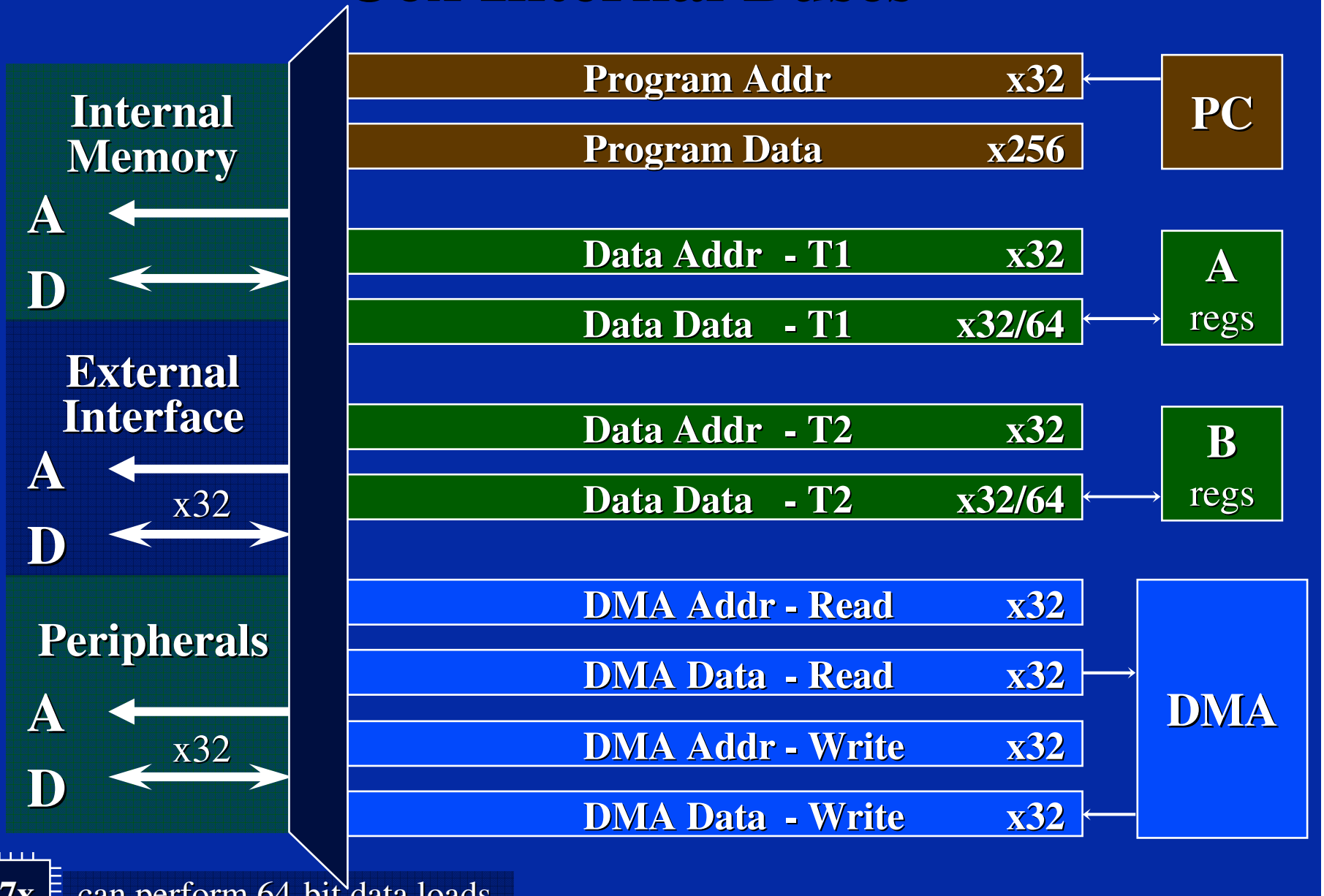


TMS320C6211/6711 Peripherals

'C6x System Block Diagram

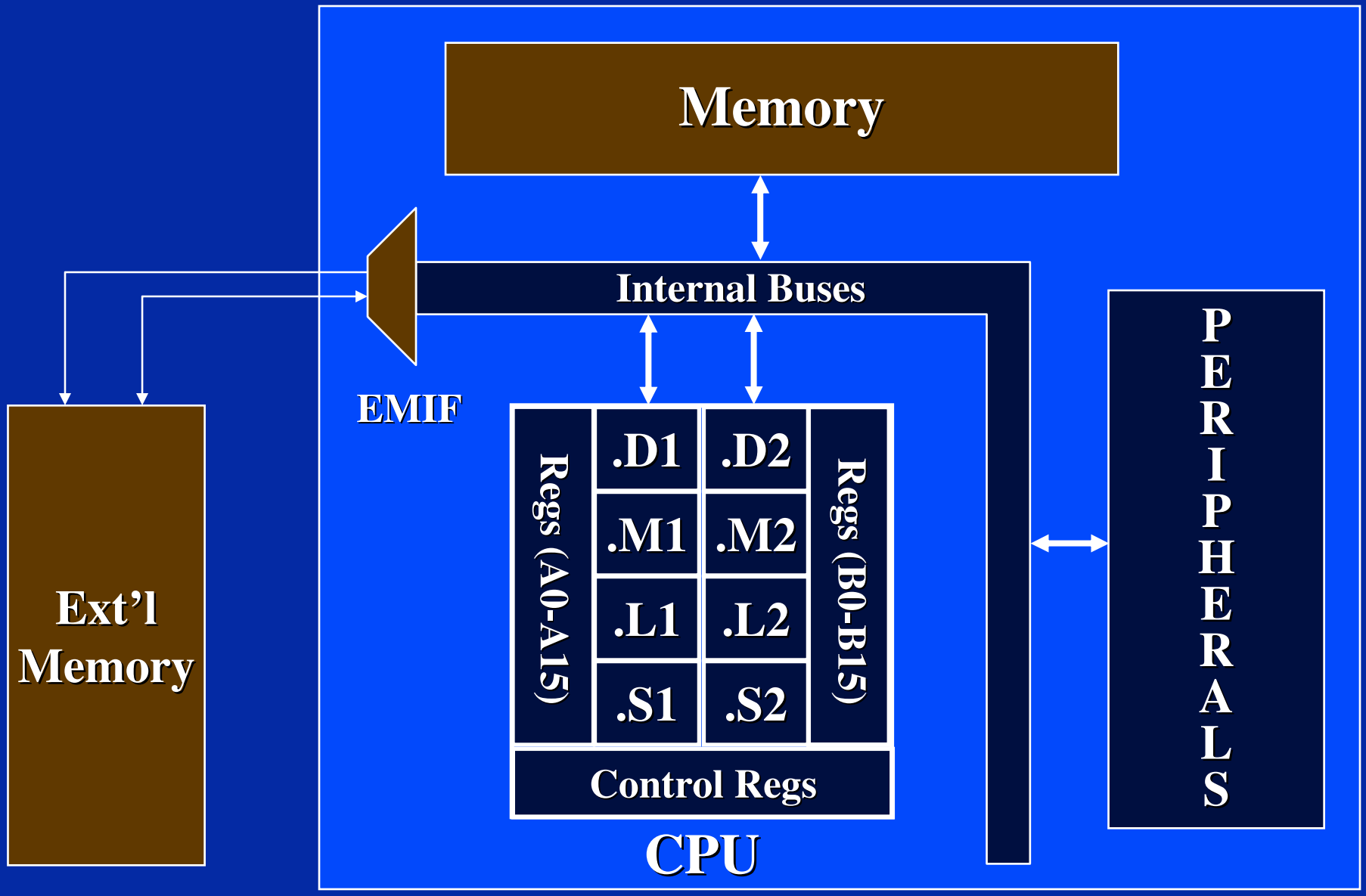


‘C6x Internal Buses



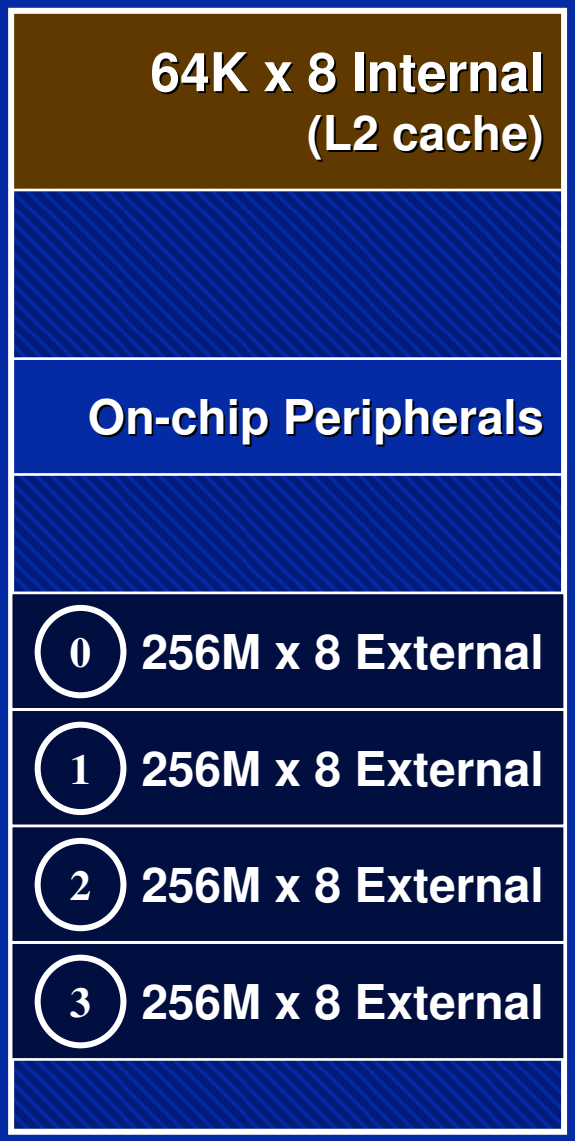
‘C67x can perform 64-bit data loads.

'C6x System Block Diagram



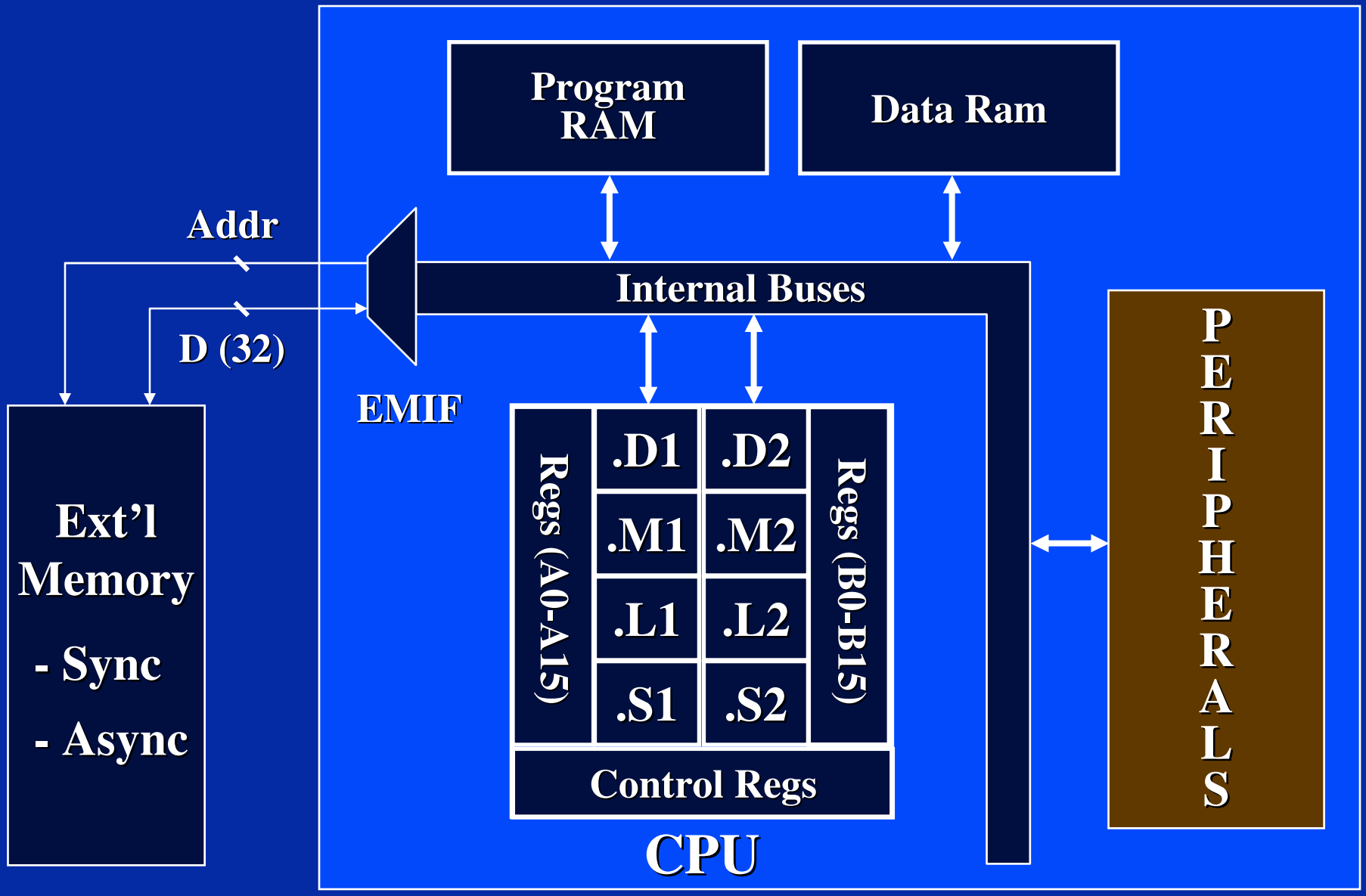
‘C6201/11 Memory Maps

‘C6202

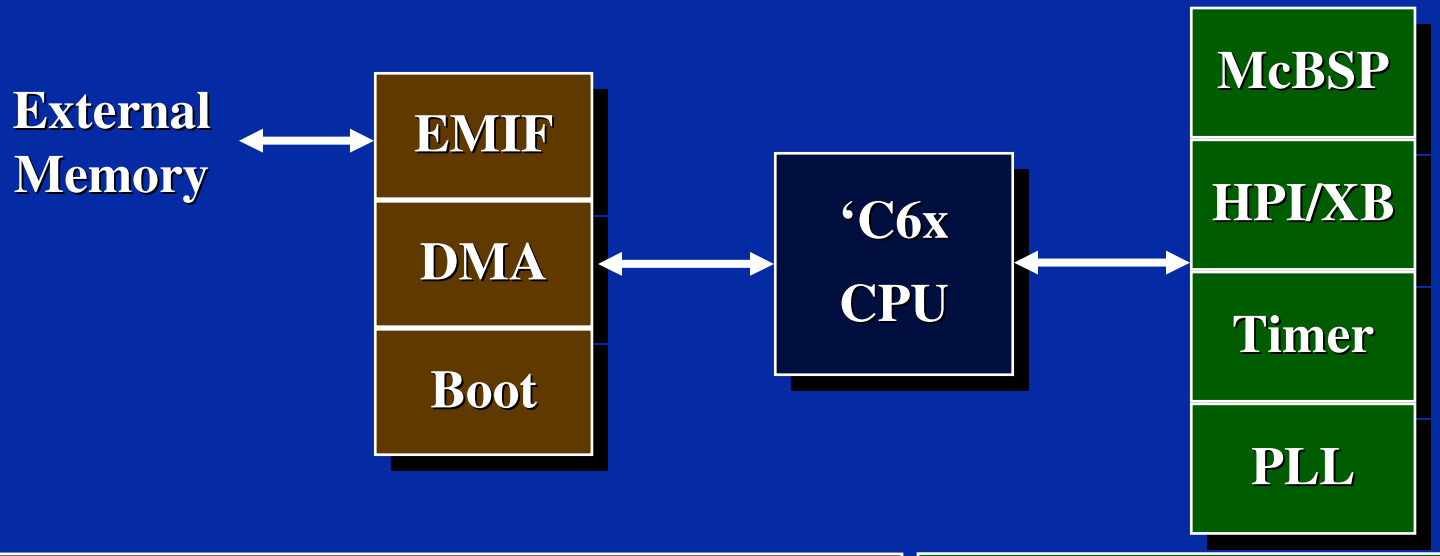


‘C6211

'C6x System Block Diagram



'C6x Peripherals



EMIF (External Memory Interface)

- Glueless access to async/sync memory
  EPROM, SRAM, SDRAM, SBSRAM

DMA/EDMA (Enhance Direct Memory Acces)

- 4/16 Channels

BOOT

- Boot from 4M external block
- Boot from HPI/XB

McBSP (Multi-Channel Buffered Serial Port)

- High speed sync serial comm
- T1/E1/MVIP interface

HPI (Host Port Interface) /Expansion Bus (XB)

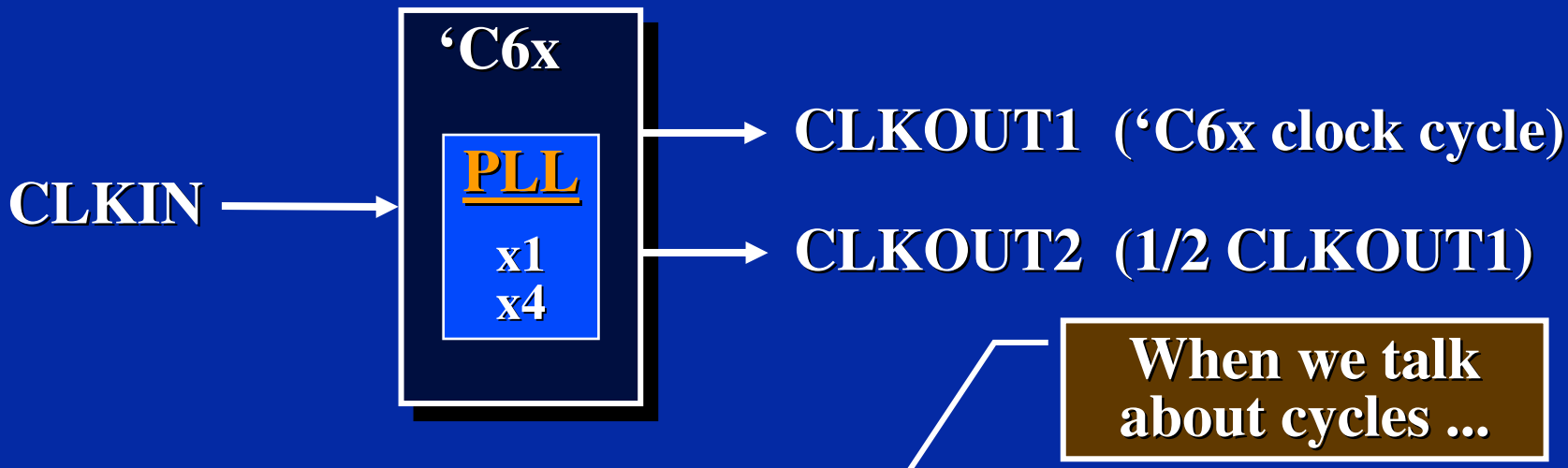
- 16/32-bit host µP access

Timer/Counters

- Two 32-bit Timer/Counters

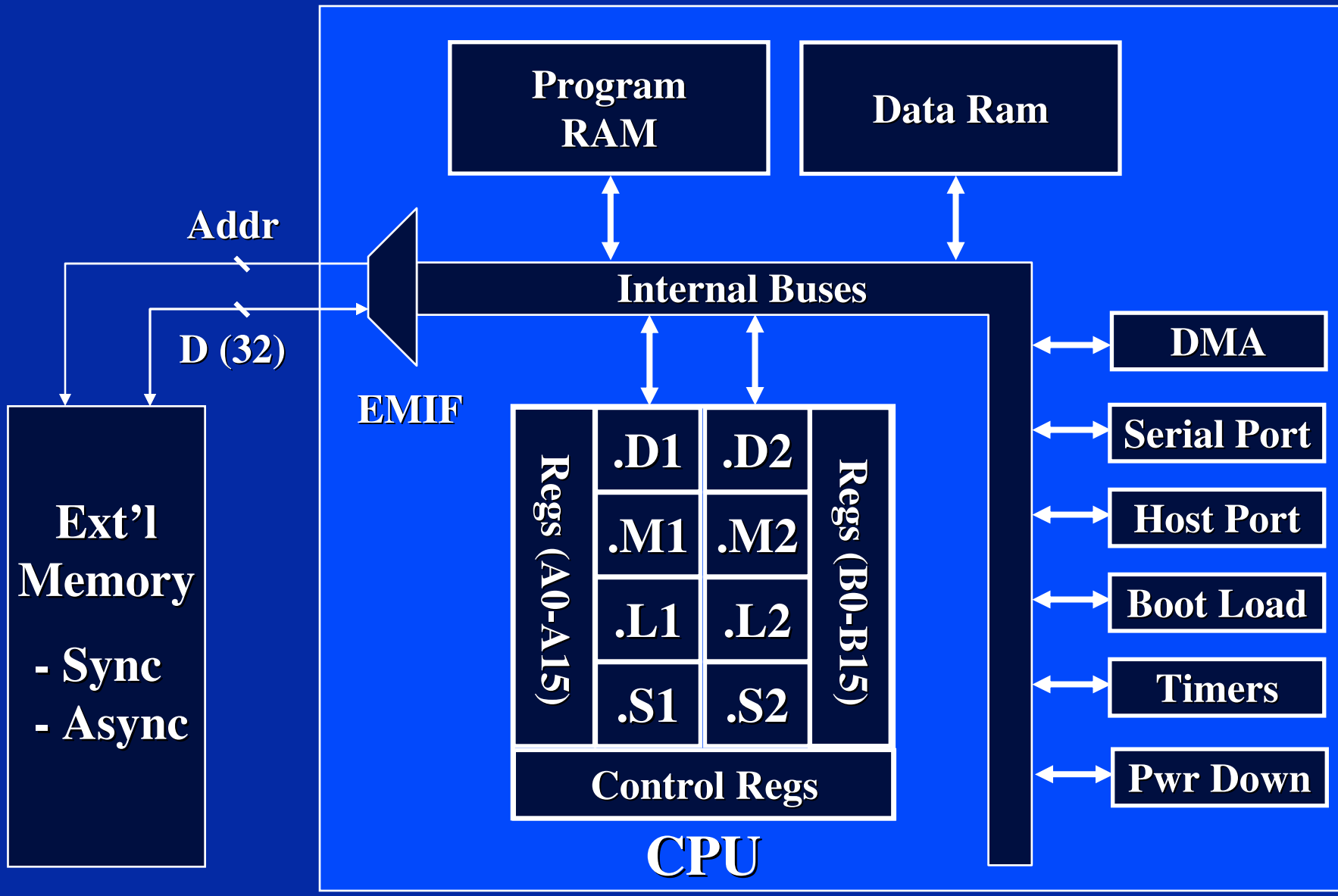
Clocking - Basic Definitions

What is a "clock cycle"?



CLKIN - MHz	PLL	CLKOUT1 - MHz	CLKOUT2 - MHz	MIPs (max)
250	x1	250 (4ns)	125	2000
200	x1	200 (5ns)	100	1600
50	x4	200	100	1600
25	x4	100 (10ns)	50	800

'C6x System Block Diagram (Final)



Internal Memory Summary

	L1 Memory		L2 Memory
	Program Memory	Data Memory	
'C6201B	**64KB** 1 blk Pgm/Cache	**64KB** 2 blks 4 banks ea	**External**
'C6701	**64KB** 1 blk Pgm/Cache	**64KB** 2 blks 8 banks ea	**External**
'C6202	**256KB** 1 blk Pgm/Cache 1 blk Mapped Pgm	**128 KB** 2 blks 4 banks ea	**External**
'C6211/C6711	**4 KB** 1 blk Cache	**4 KB** 1 blk Cache	**64 KB** 4 blk Mapped Cache

TMS320C62x DSP Generation Parametric Table

TMS320C64x DSP Generation Parametric Table

TMS320C67x DSP Generation Parametric Table

‘C6000 Device Summary

Device	MIPS	MHz	Kbytes	pins	mm	W	$	Periphs
6201B	1600	200	128	352	27	1.9	80-110	D2H
6202	2000	250	384	352	27	1.9	120-150	D3X
6211	1200	150	72	256	27	1.5	20-40	E2H

TMS320	MFLOPS	MHz	Kbytes	pins	mm	W	$	Periphs
6701	1000	167	128	352	35	1.9	170-200	D2H
6711	600	100	72	256	27	0.9	20-40	E2H

Peripherals Legend:
D,E: DMA,EDMA
2,3: # of McBSPs
H,X: HPI, XBUS

TMS320C62x DSP Generation Parametric Table

TMS320C64x DSP Generation Parametric Table

TMS320C67x DSP Generation Parametric Table

'C6000 History

6201 *r1*	1Q97 coincident 'C6x architectural announcement. Sample CPU core, minimal peripherals.
6201 *r2*	4Q97. Full production, with peripherals.
6201B	4Q98. Power reduced, .18 micron silicon, double ports into internal data memory.
6701	3Q98. Pin-for-pin compatible floating-point version of 'C6201. 1GFLOP (@ 167MHz) performance.
6202	2Q99. 2000 MIPS @ 250MHz. 2-3x 6201 on-chip memory.
	Replaced HPI with Expansion Bus (32-bit HPI + more).
6211	3Q99. 2 cents per MIPS! 1200MIPS @ 150MHz as low as $25. Double-level cache, enhanced DMA.
6711	Announced 3/1/99. 6701 floating-point CPU with 6211-like memory/peripherals. Volume pricing under $20.

‘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

Device Summary Table

Device	Int Mem	Ext Mem	Peripherals
6201/6701	64K Data 16K Instr	3 x 16M 1 x 4M	DMA 2 McBSP HPI (16-bit) 2 Timer/Counters (32-bit)
6202	128K Data 48K Instr	3 x 16M 1 x 4M 4 x 256M ----	DMA 2 McBSP XBus (32-bit) 2 Timer/Counters (32-bit)
6211	4K Data Cache 4K Prog Cache 64K RAM/Cache	4 x 256M	EDMA 2 McBSP HPI (16-bit) 2 Timer/Counters (32-bit)

C64x Quad 8 x 8 Multiply