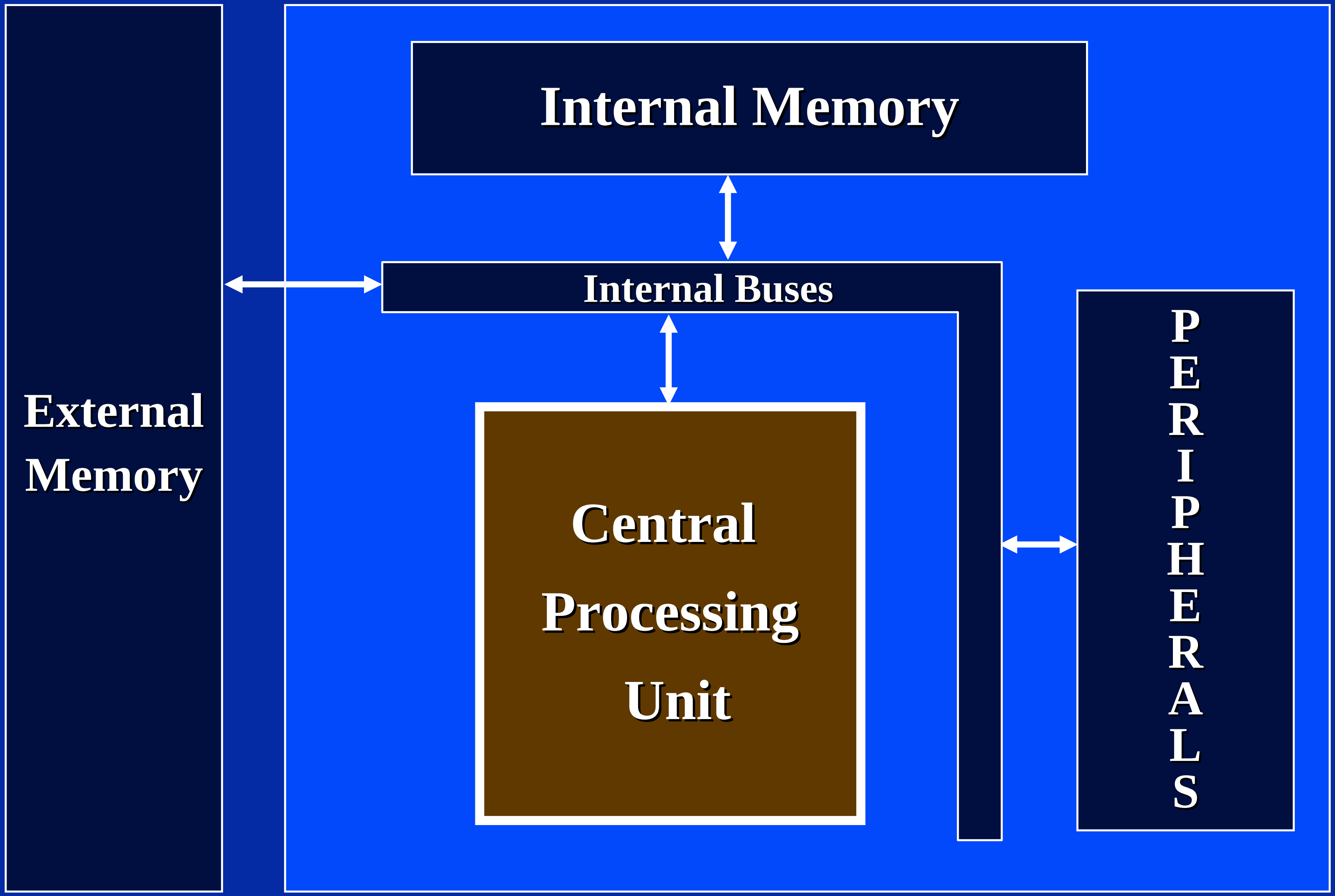
Chapter 1

TMS320C6000 Architectural Overview

Learning Objectives

- Describe C6000 CPU architecture.
- Introduce some basic instructions.
- Describe the C6000 memory map.

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)

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

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

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)

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

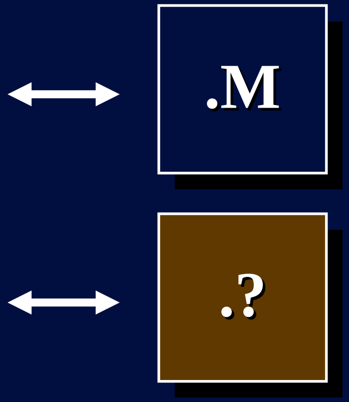
The . M unit performs multiplications in hardware

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.

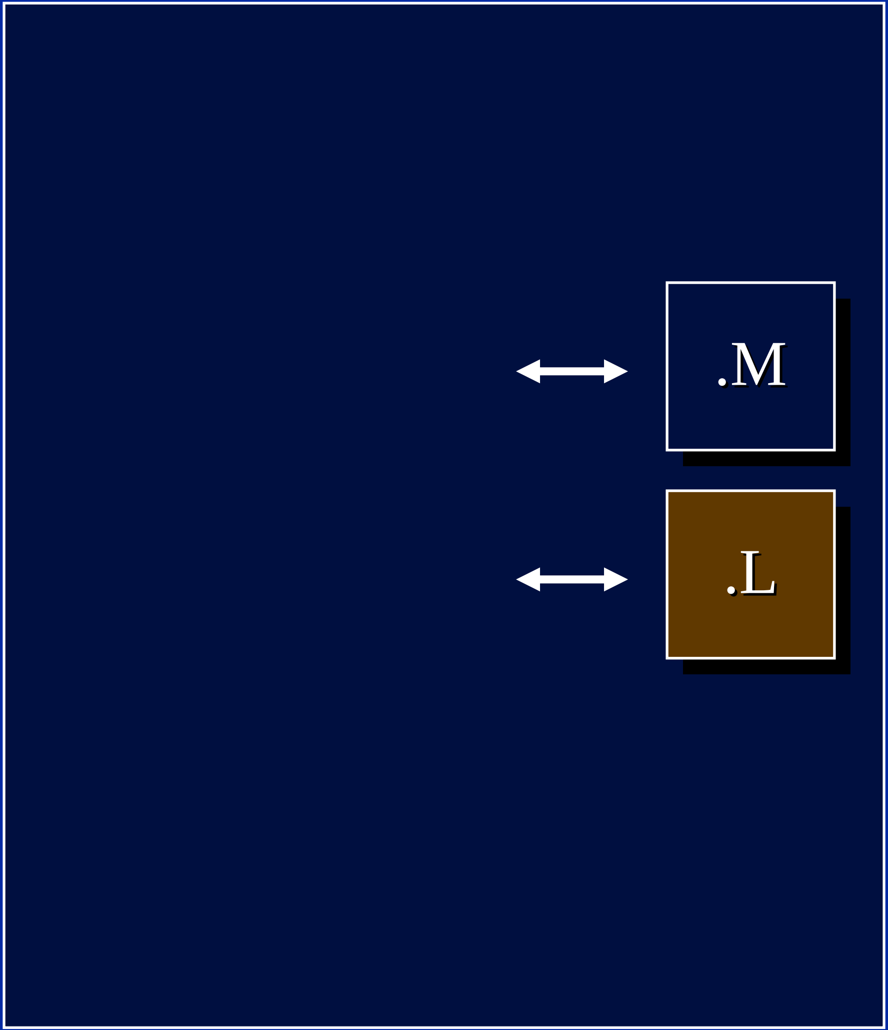
Addition (.?)



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

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

Add (.L unit)

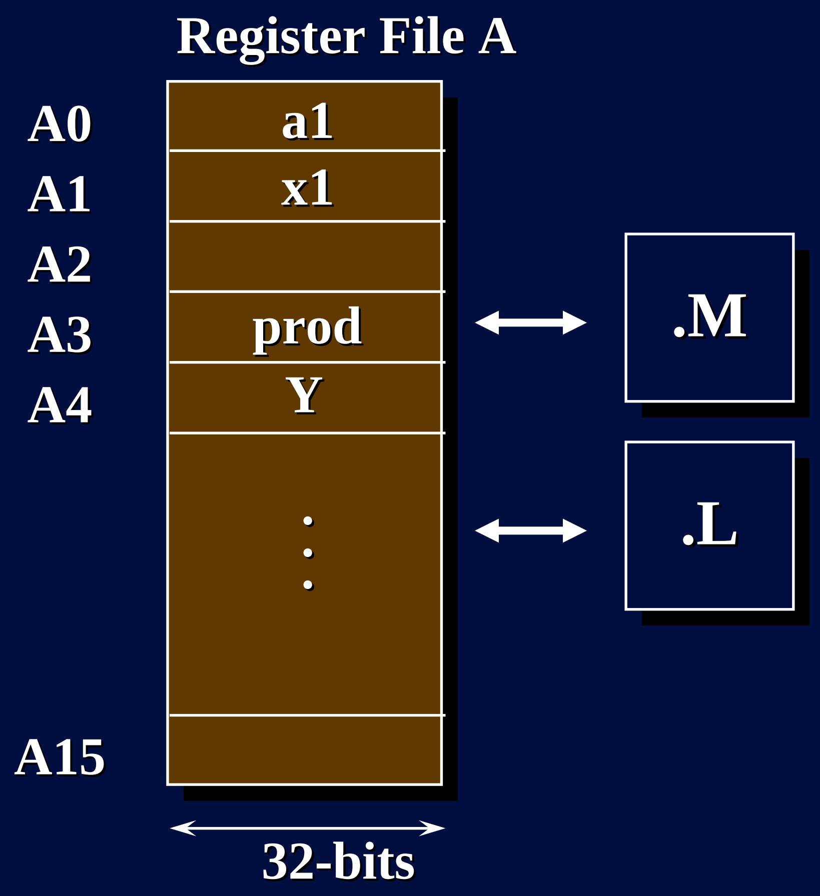


$$Y = \sum_{n=1}^{40} a_n * x_n$$

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

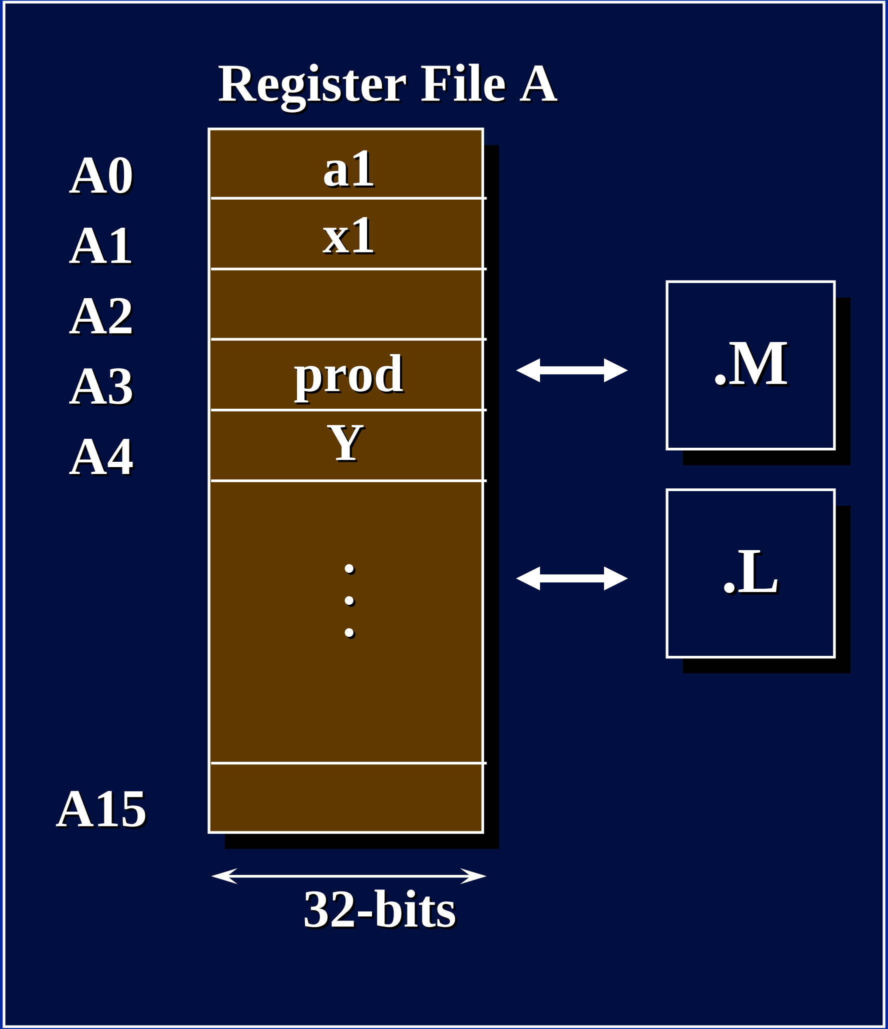


$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
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

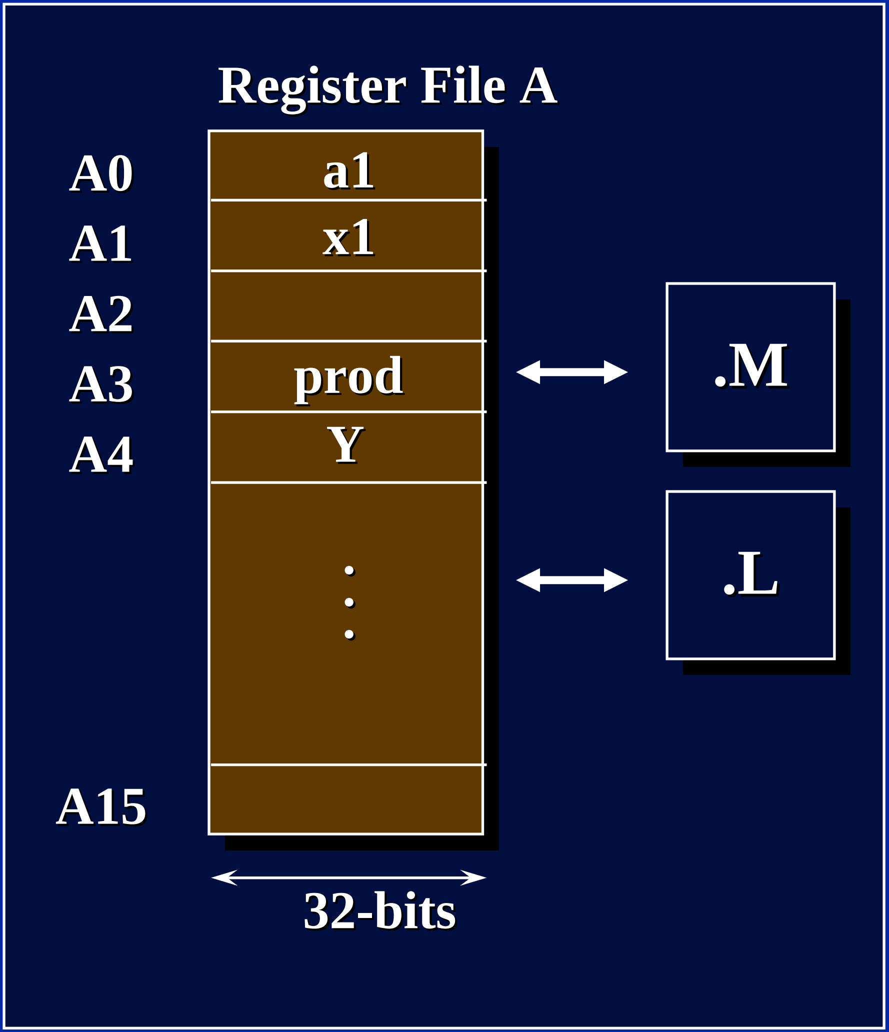


$$Y = \sum_{n=1}^{40} a_n * x_n$$

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

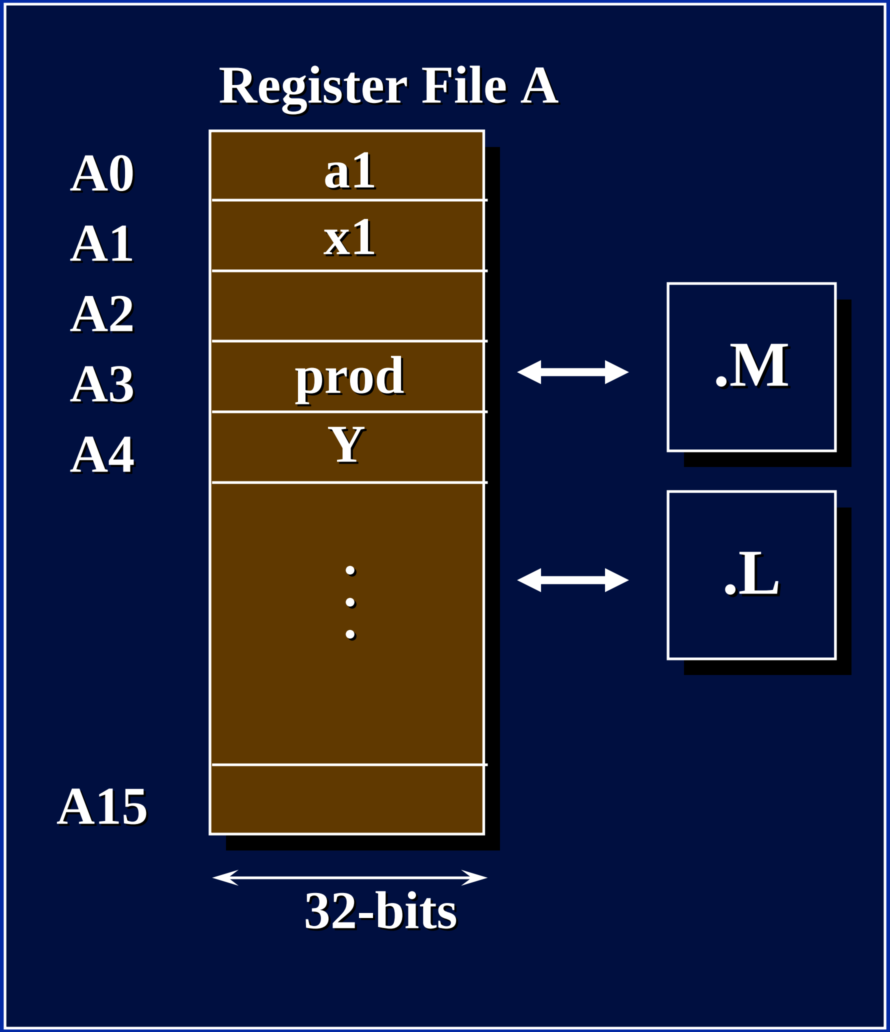


$$Y = \sum_{n=1}^{40} a_n * x_n$$

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

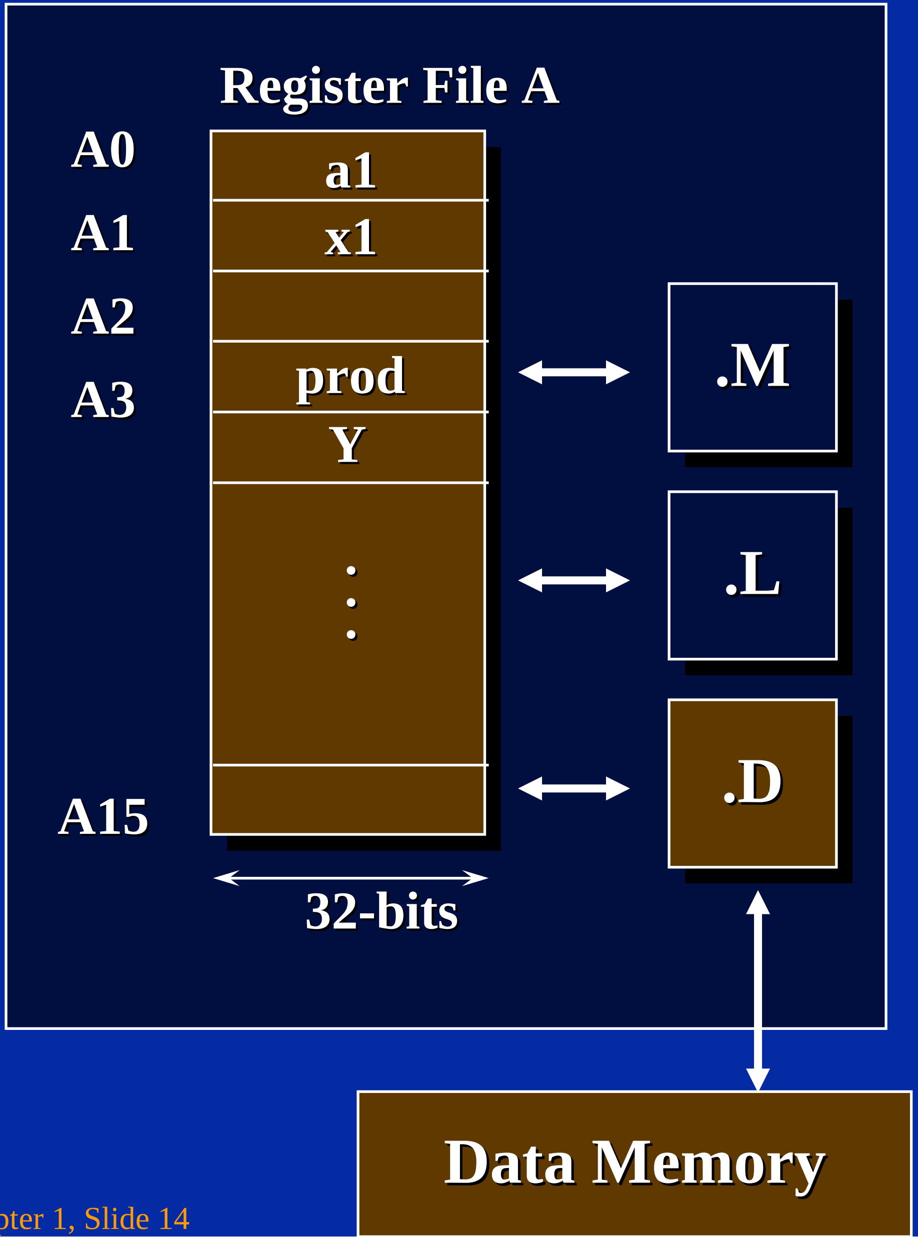
Register File A contains 16 registers (A0 -A15) which are 32-bits wide.

Data loading



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

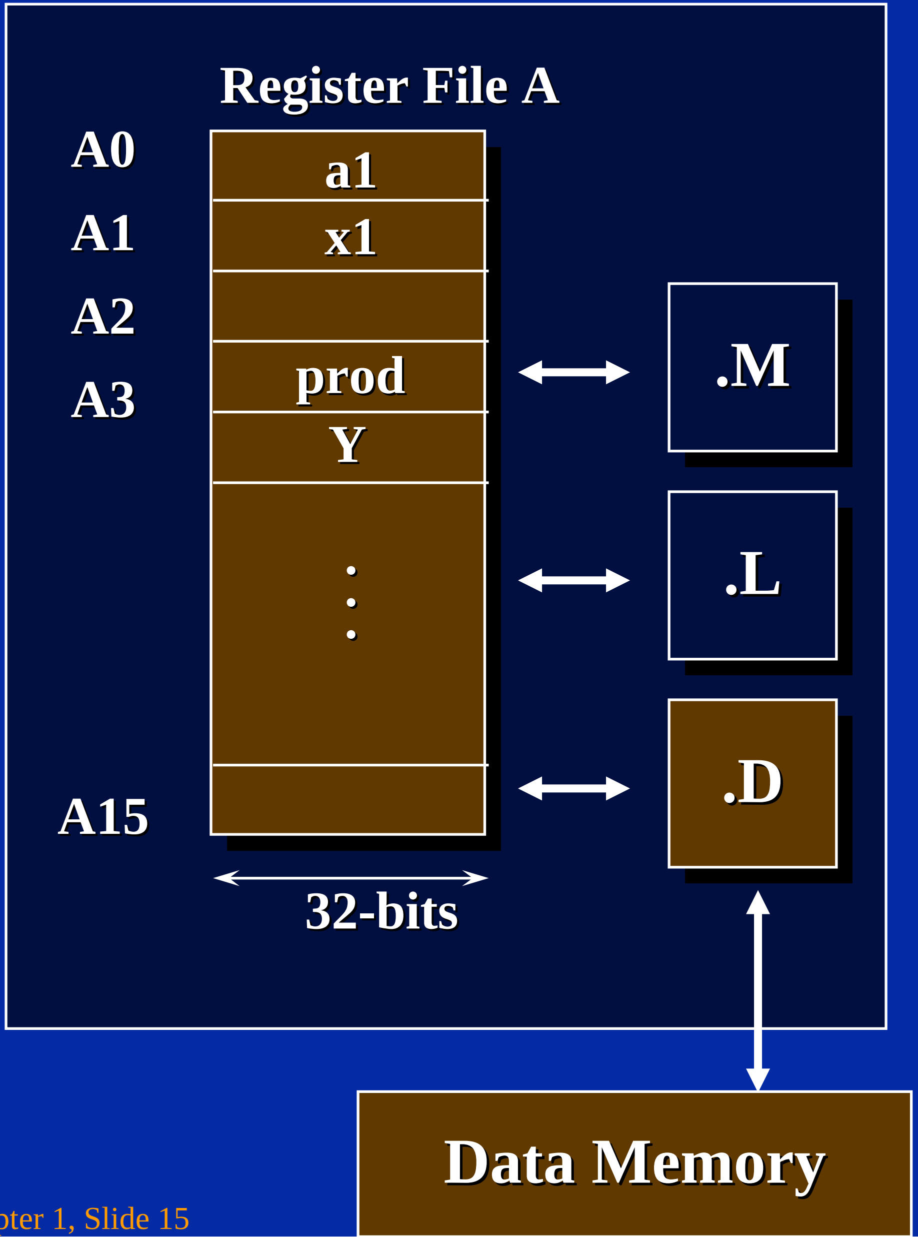
Load Unit ".D"



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

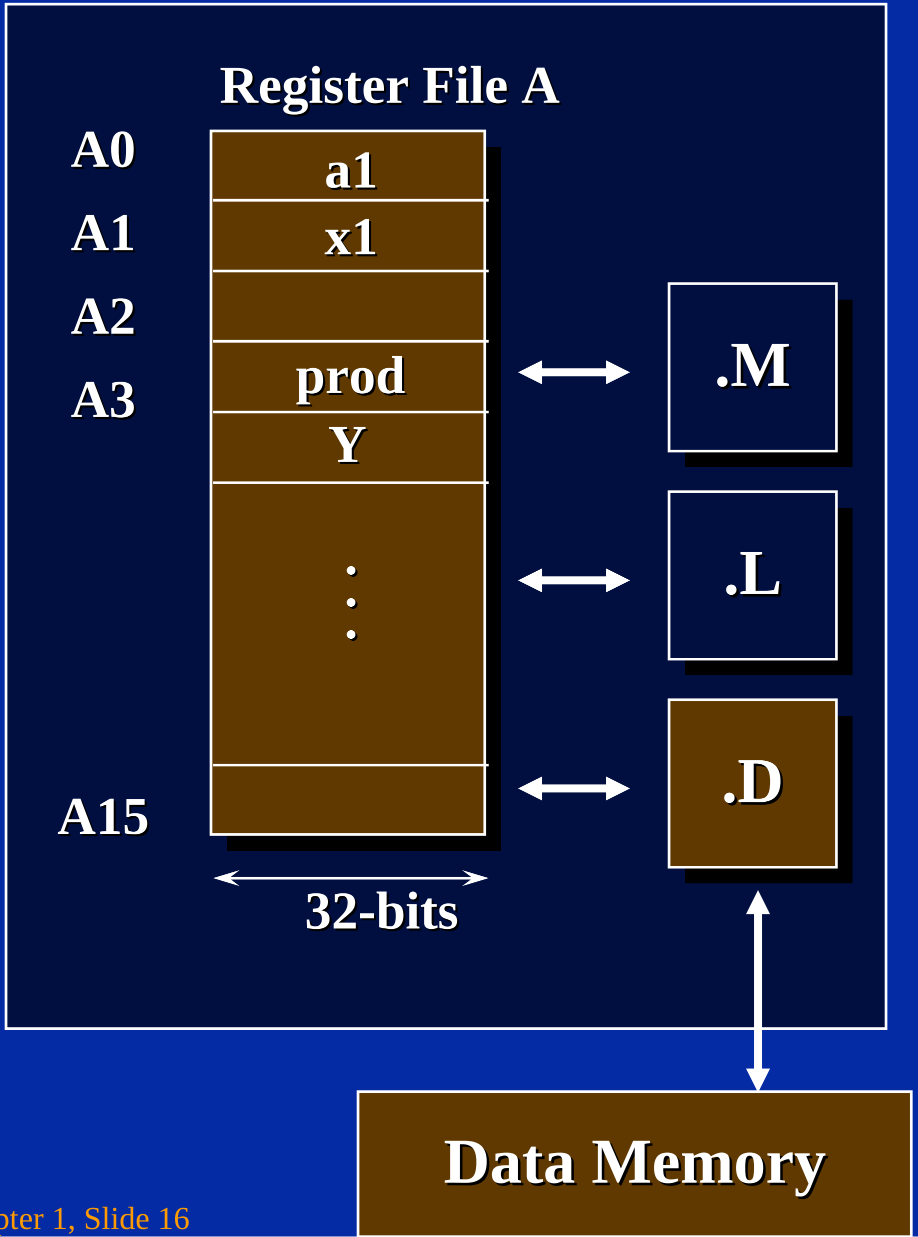
A: The operands are loaded into the registers by loading them from the memory using the .D unit.

Load Unit ".D"



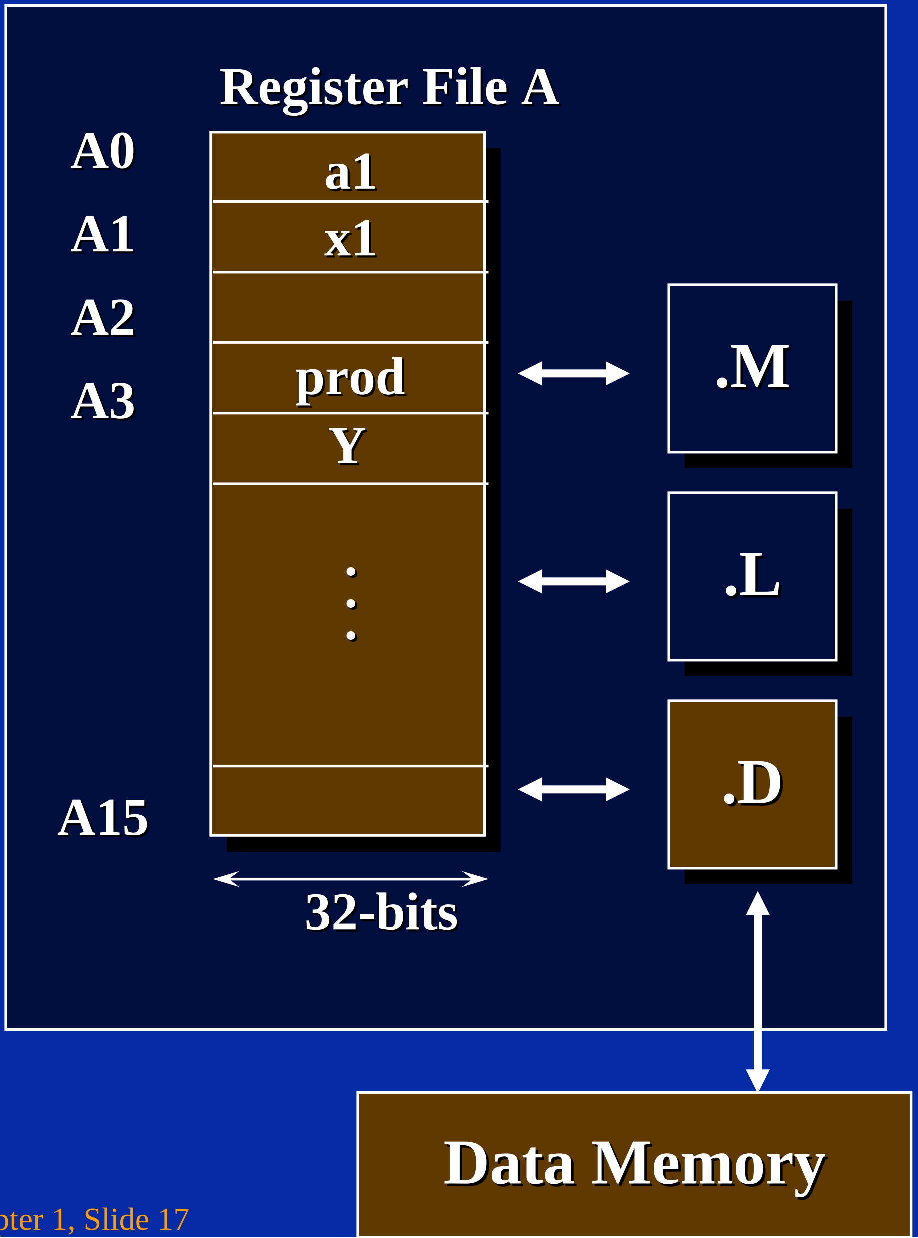
It is worth noting at this stage that the only way to access memory is through the .D unit.

Load Instruction



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

Load Instructions (LDB, LDH,LDW,LDDW)



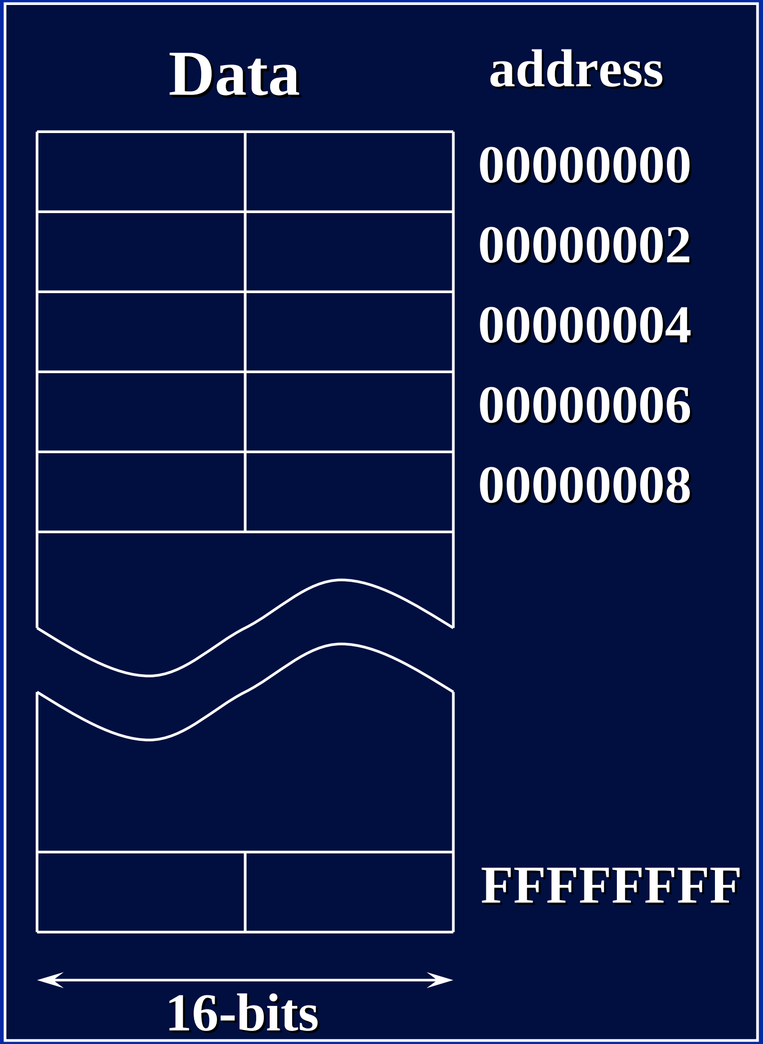
Q: Which instruction(s) can be used for loading operands from the memory to the registers?

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:

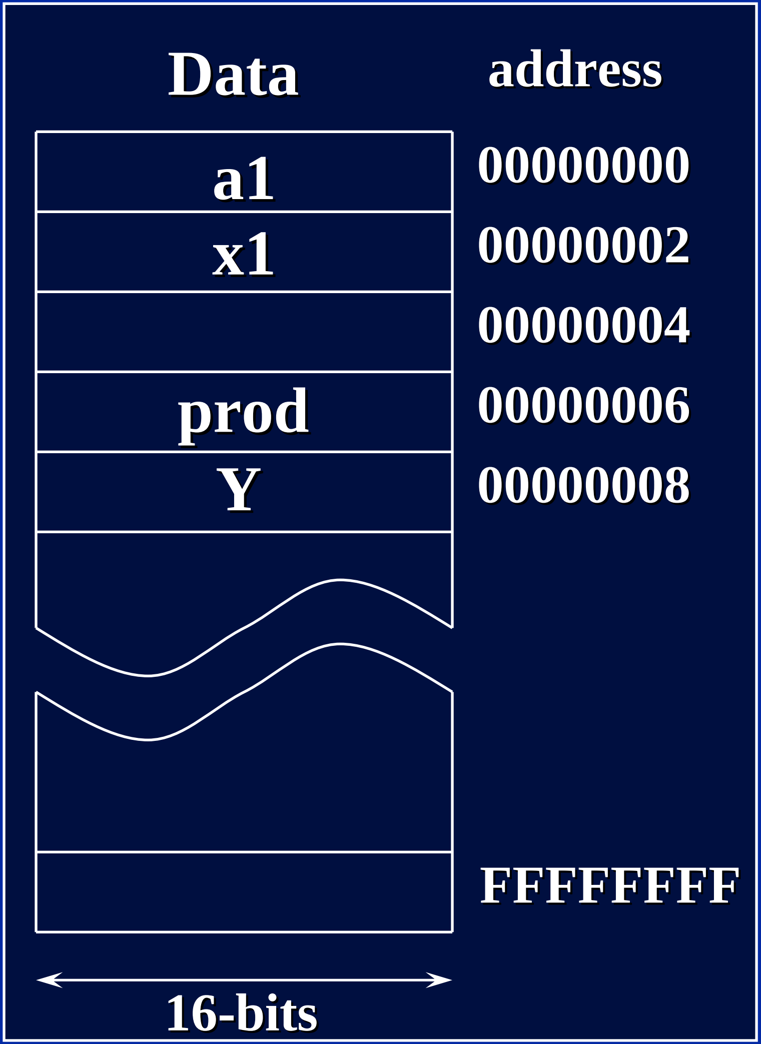
```
LD   *Rn,Rm
```

Where:

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

and

Rm is the destination register.

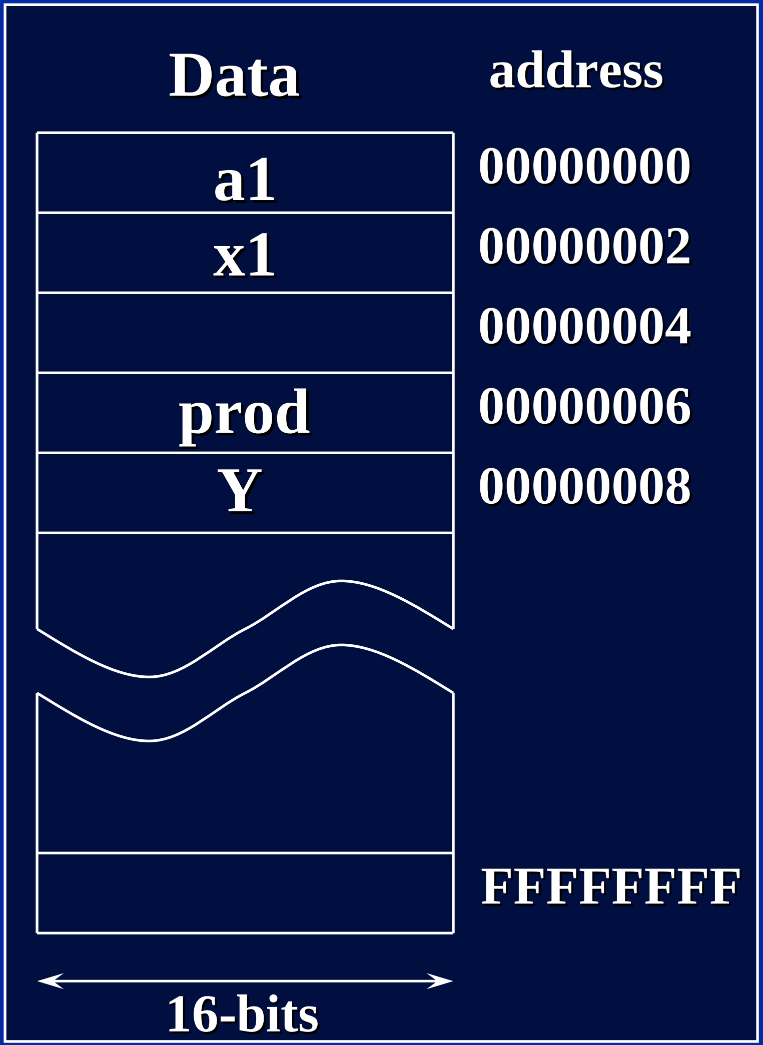


Using the Load Instructions

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

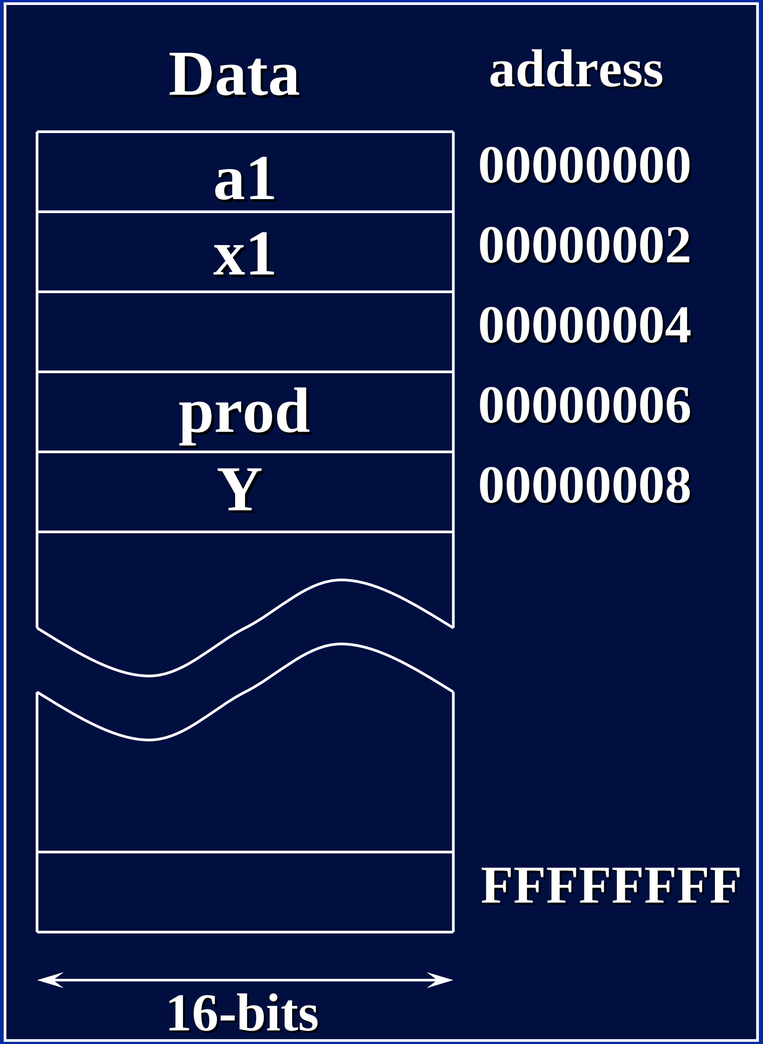
The syntax for the load instruction is:

```
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:

$$\boxed{\text{LD} \quad \text{*Rn,Rm}}$$

Example:

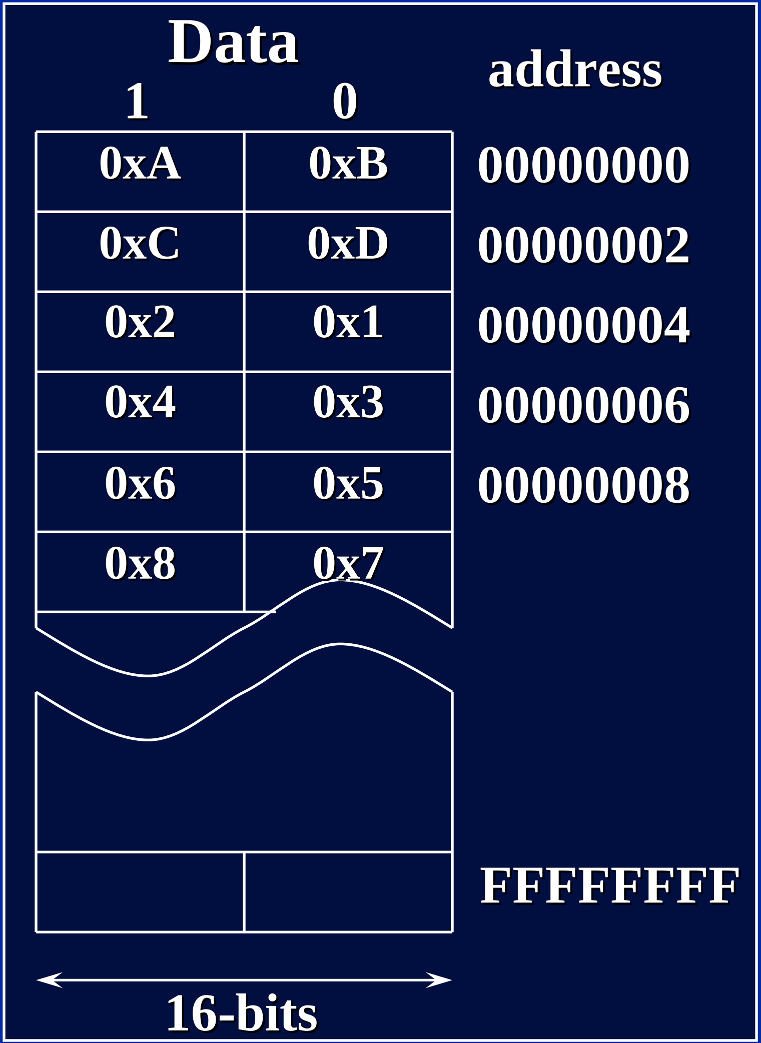
If we assume that A5 = 0x4 then:

(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



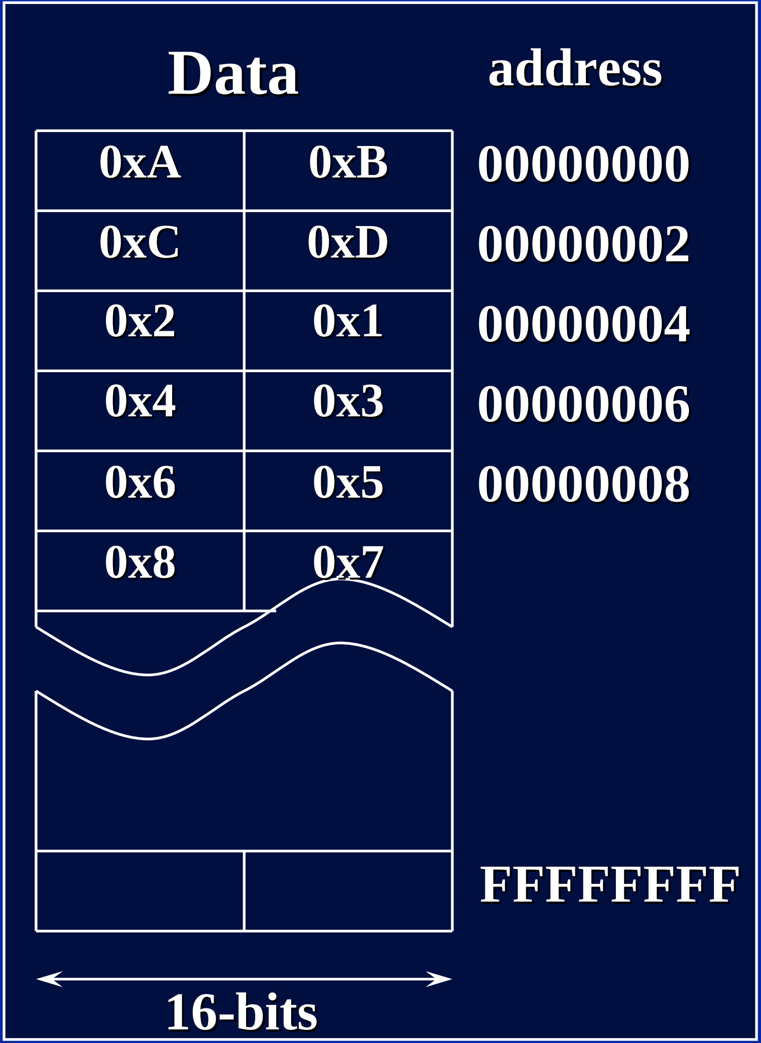
Using the Load Instructions

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).

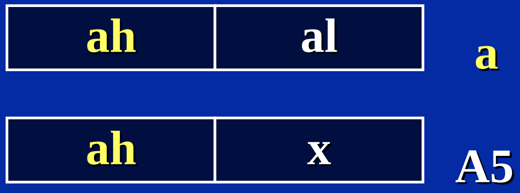
Loading the Pointer Rn

- 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)
```

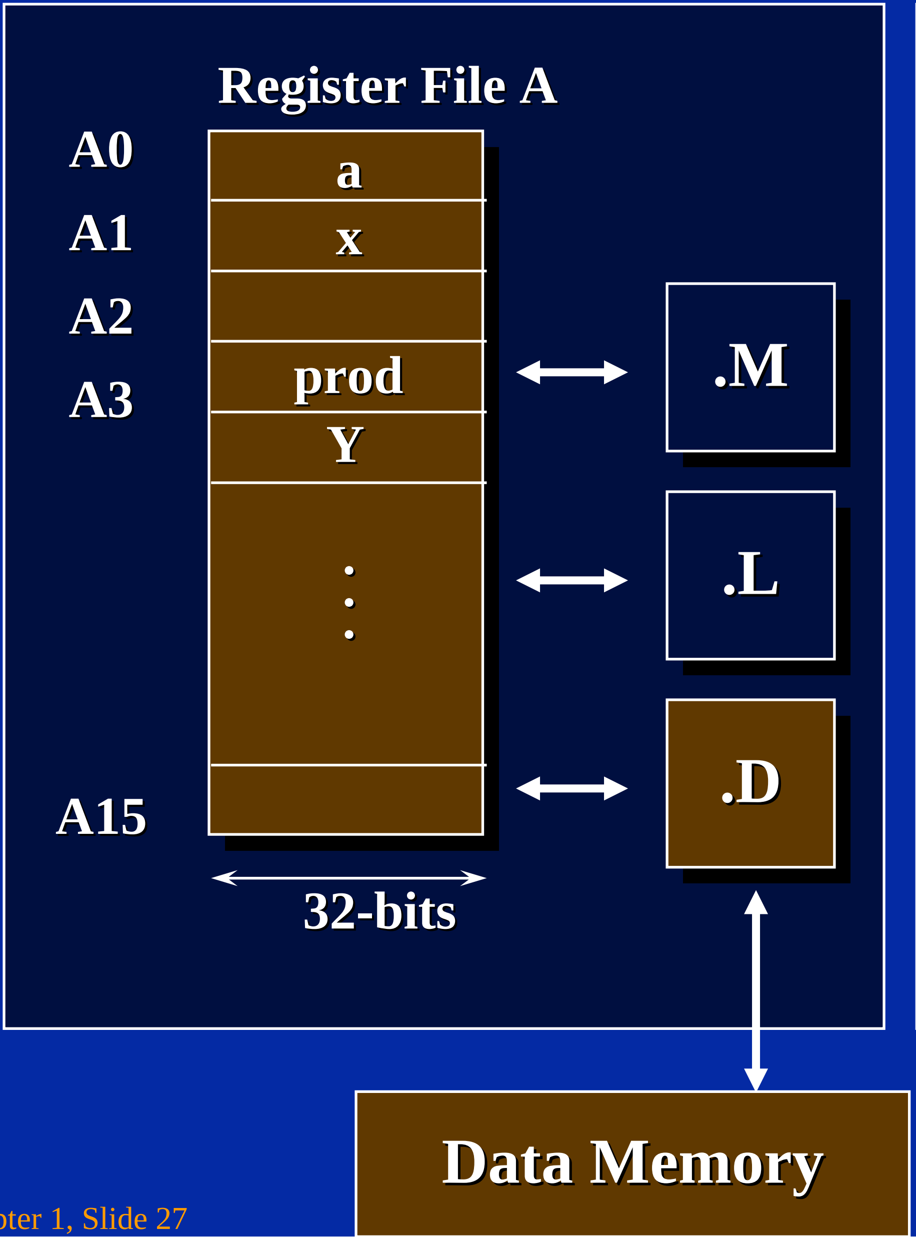
```
MVKH            0x1234FABC, A5
A5 = 0x1234FABC  ; OK
```

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.

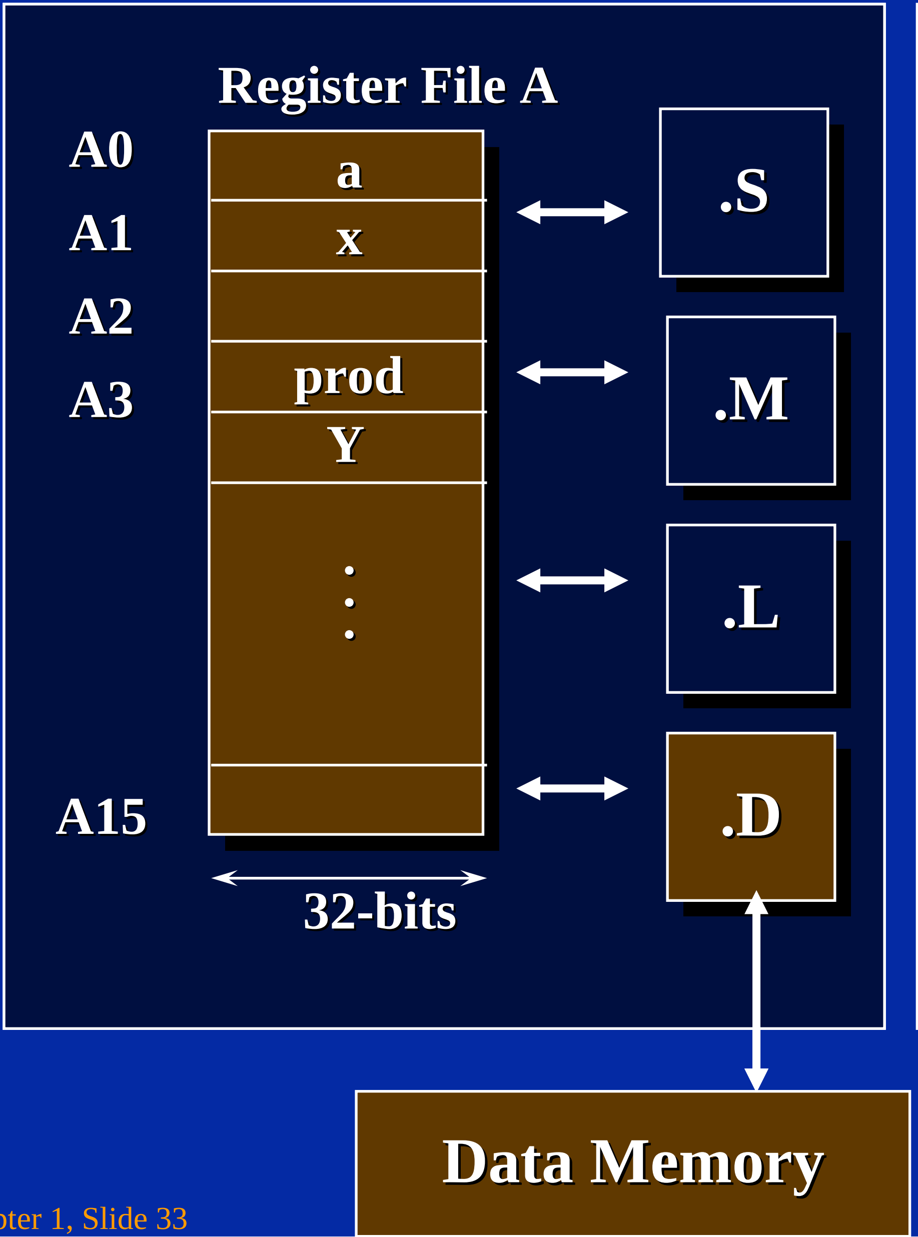
1. Create a label to branch to

```
        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.

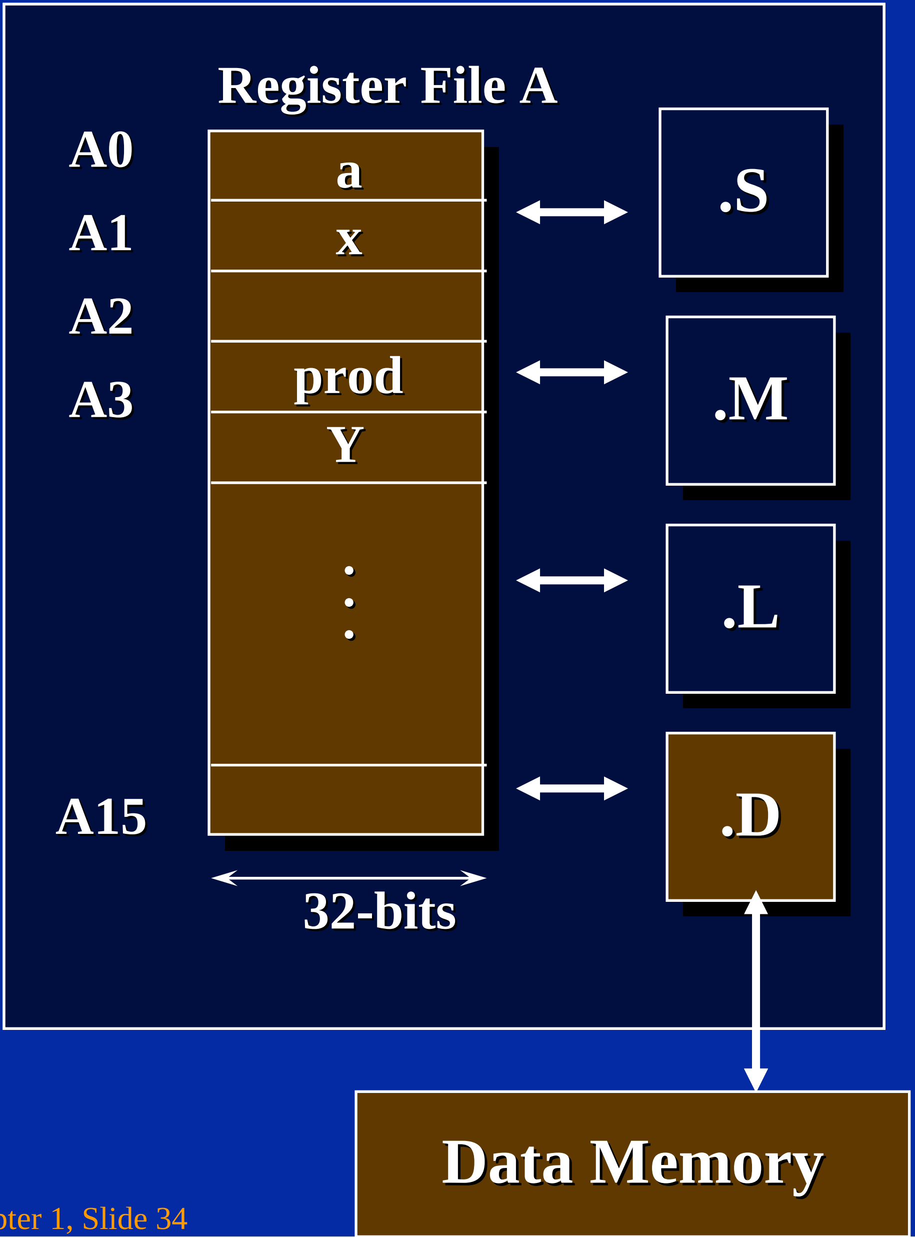
```
            MVKL            pt1, A5
            MVKH            pt1, A5

            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?



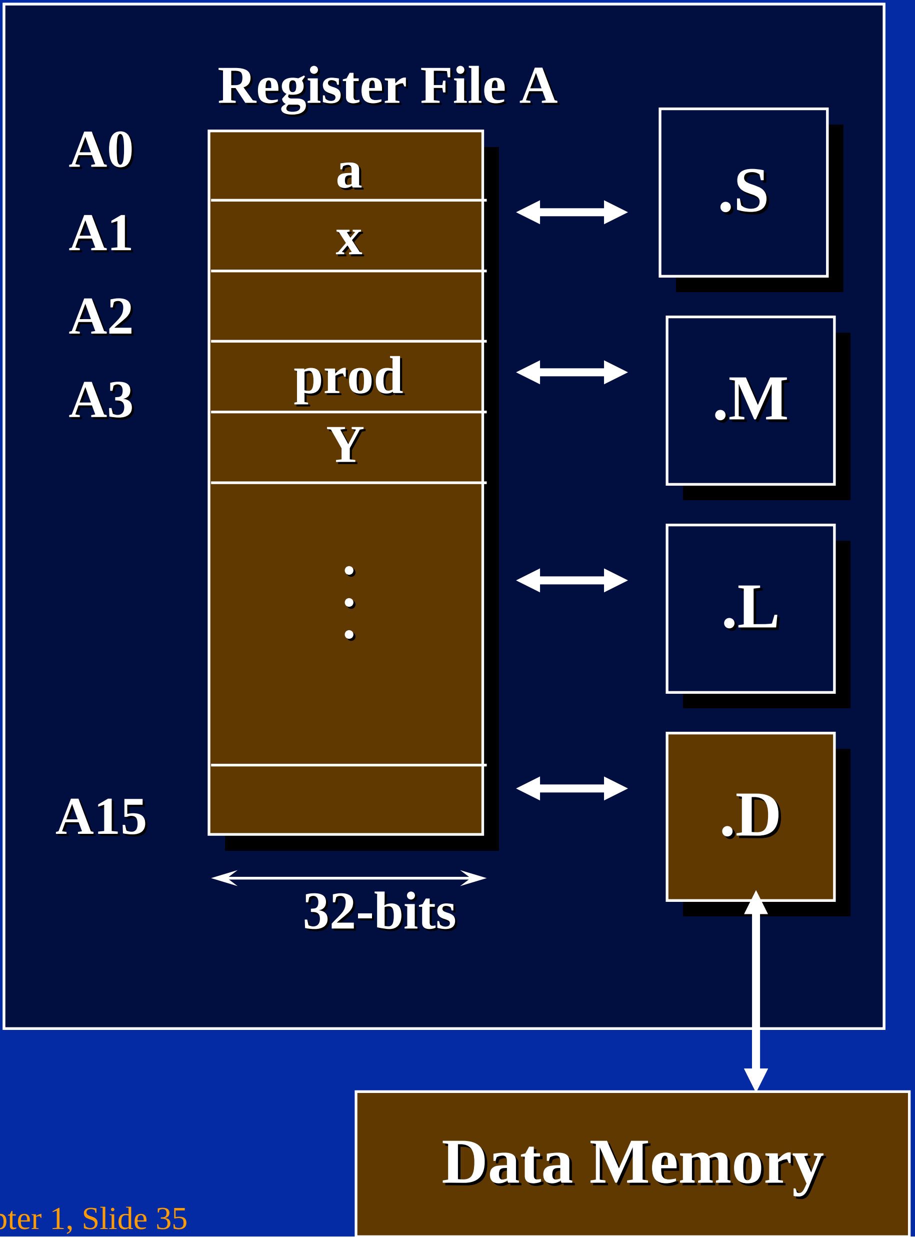
```
          MVKL        pt1, A5
          MVKH        pt1, A5

          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
```

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
```

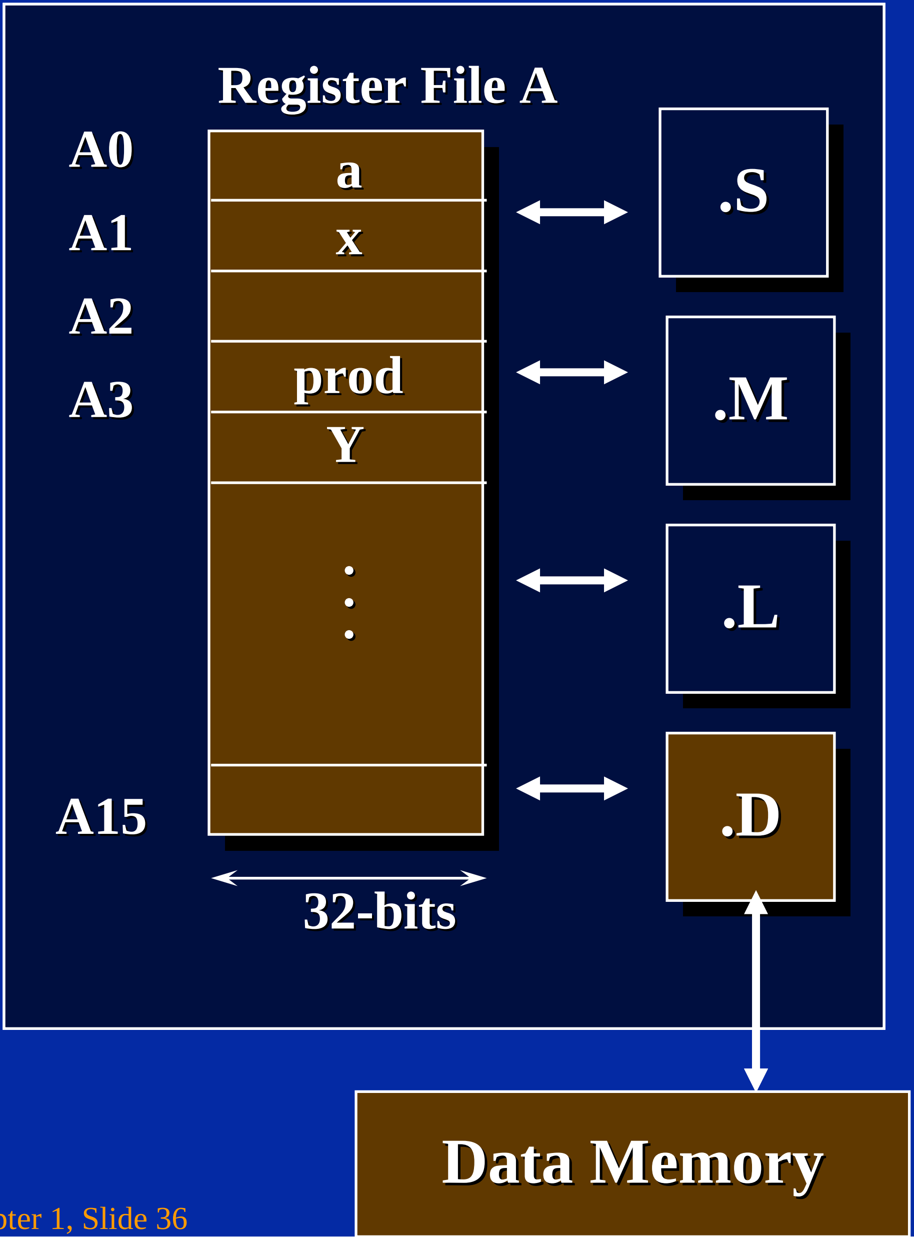
3. Create a loop counter.



```
                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

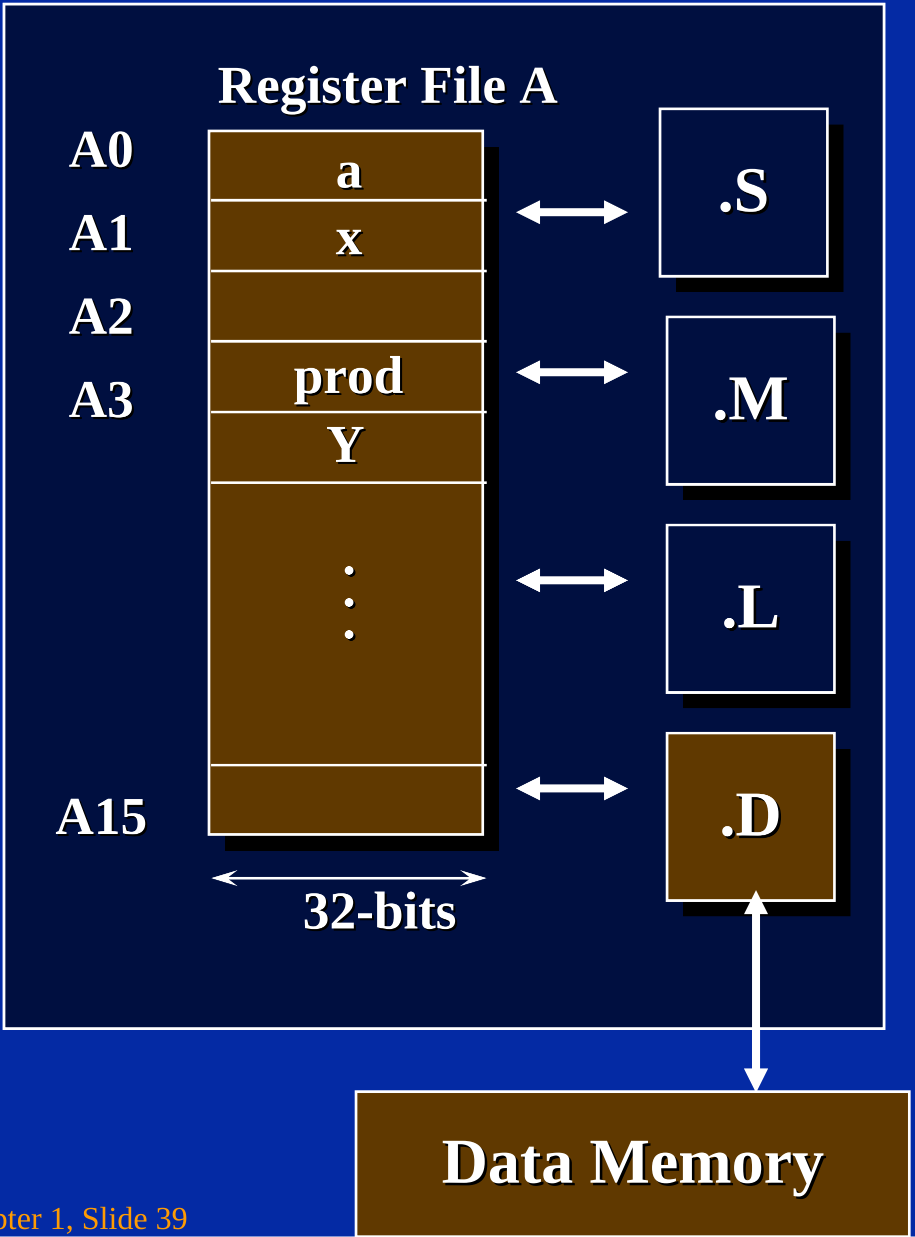
- The condition can be inverted by adding the exclamation symbol “!” as follows:

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

e.g.

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

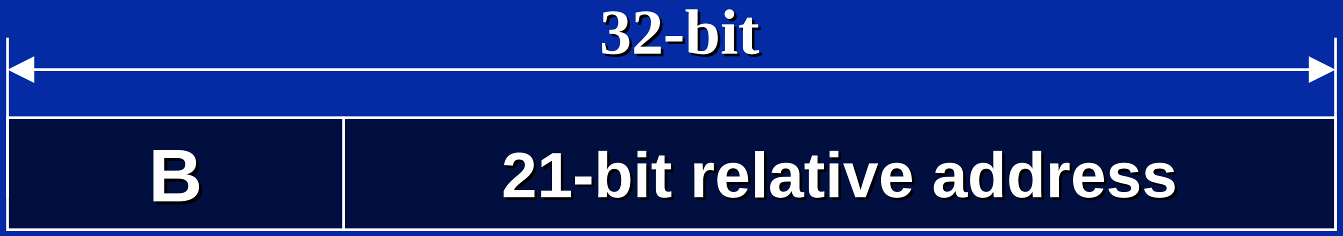
5. Make the branch conditional



```
            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)

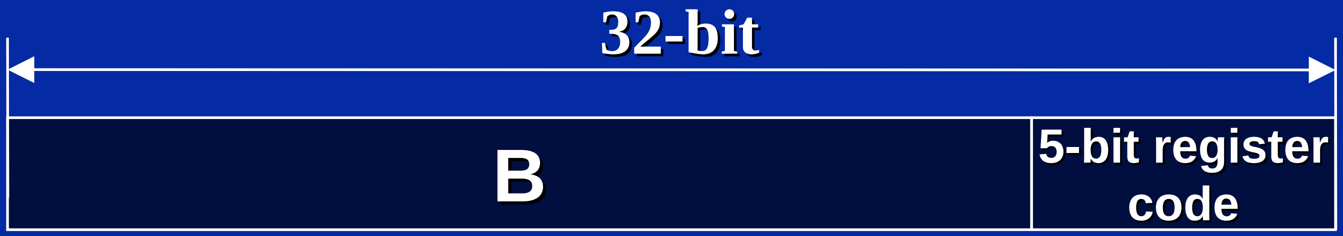
- With this processor all the instructions are encoded in a 32-bit.
- Therefore the label must have a dynamic range of less than 32-bit as the instruction B has to be coded.



- 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:

$a_0*x_0 + a_0*x_0 + a_0*x_0 + \dots + a_0*x_0$

However, we would like to perform:

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

```
            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:

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

```
            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
            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.

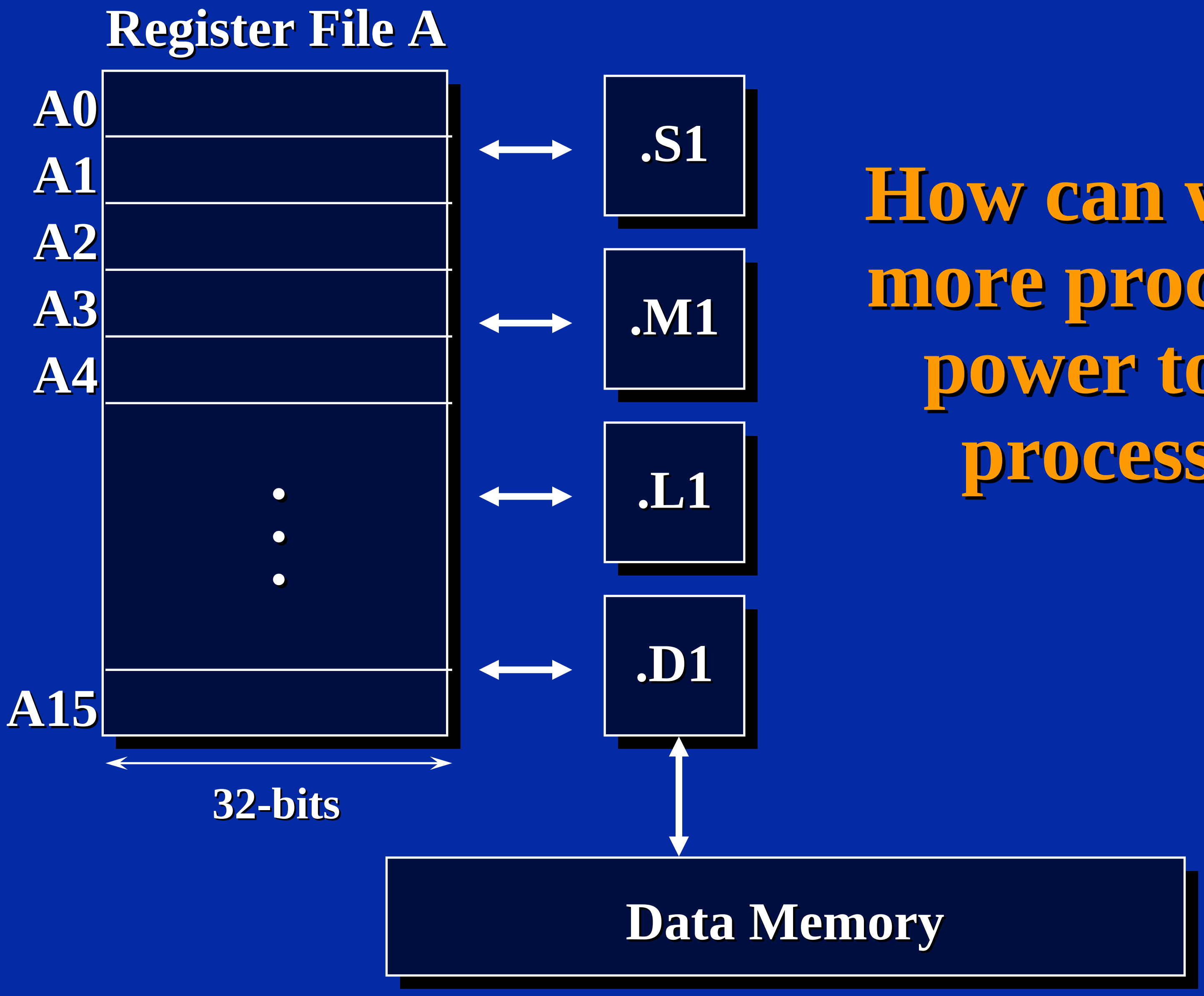
```
                MVKL   .S2   pt1, A5
                MVKH   .S2   pt1, A5

                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.

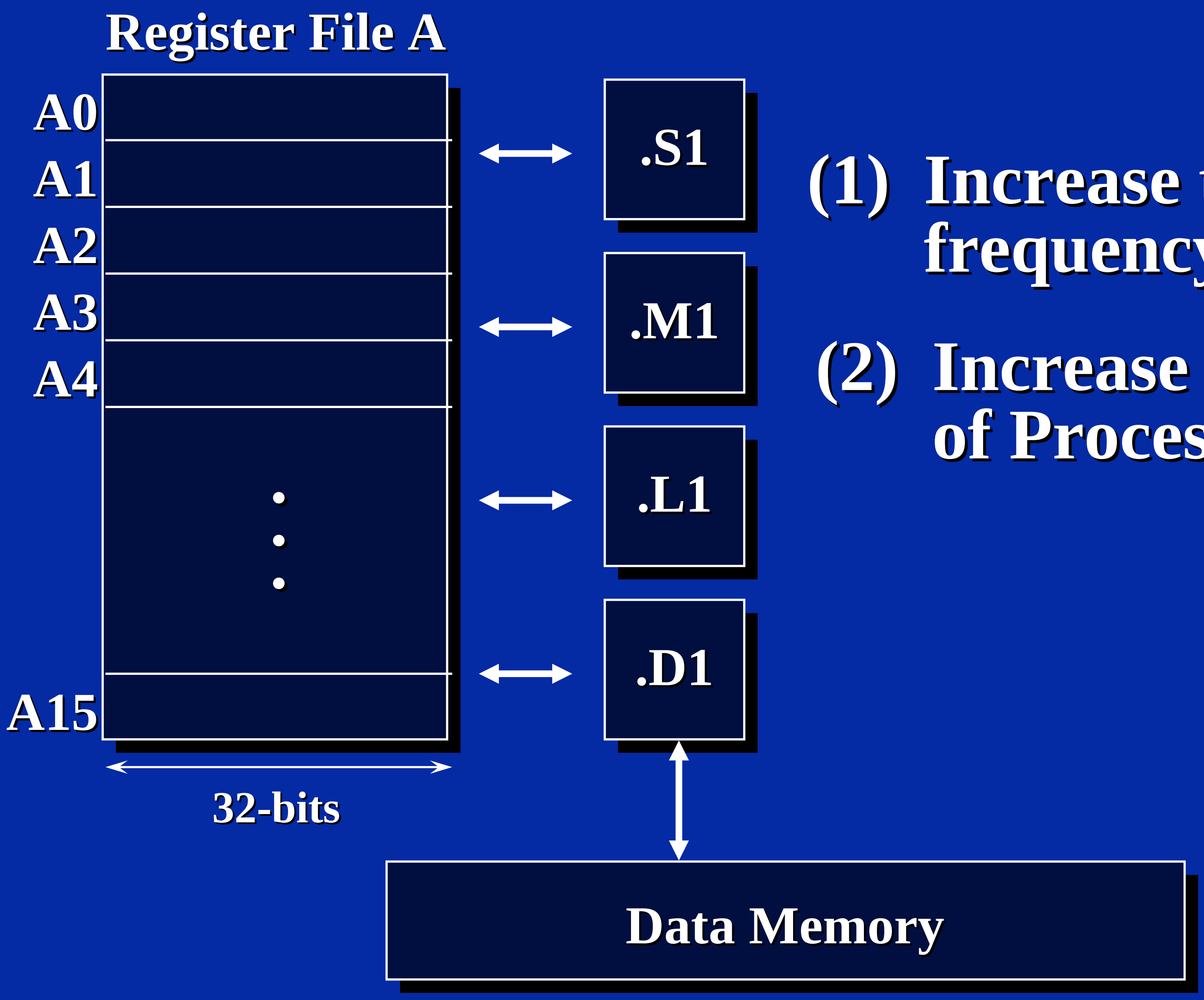
```
                MVKL   .S2   pt1, A5
                MVKH   .S2   pt1, A5

                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?

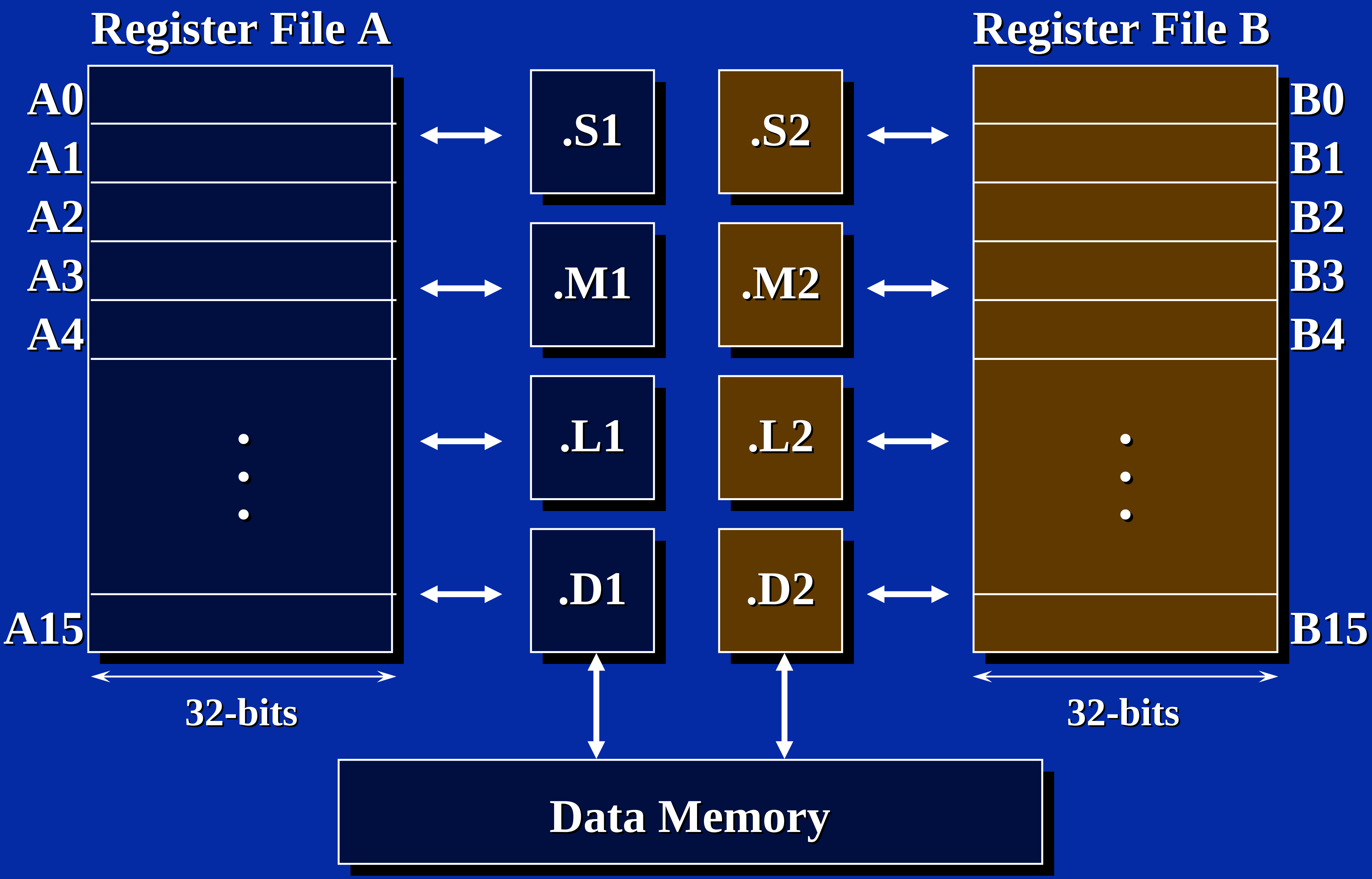
Increasing the processing power!



(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)



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
```

TMS320C6711 Instruction Set

'C6711 Instruction Set (by category)

Arithmetic	Logical	Data Mgmt	Program Ctrl	Bit Mgmt
ABS	AND	LDB/H/W	B	CLR
ADD	CMPEQ	MV	IDLE	EXT
ADDA	CMPGT	MVC	NOP	LMBD
ADDK	CMPLT	MVK		NORM
ADD2	NOT	MVKL		SET
MPY	OR	MVKH		
MPYH	SHL	MVKLH		
NEG	SHR	STB/H/W		
SMPY	SSHL			
SMPYH	XOR			
SADD				
SAT				
SSUB				
SUB				
SUBA				
SUBC				
SUB2				
ZERO				

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

'C6711 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

‘C67x

.M Unit	
MPYSP	MPYI
MPYDP	MPYID

.D Unit	
ADDAD	LDDW

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

TMS320C6711 Memory Map

‘C6711 Memory Map

Byte Address	
0000_0000	64K x 8 Internal (L2 cache)
	▨
0180_0000	On-chip Peripherals
	▨
8000_0000	0 256M x 8 External
9000_0000	1 256M x 8 External
A000_0000	2 256M x 8 External
B000_0000	3 256M x 8 External
FFFF_FFFF	▨

External Memory

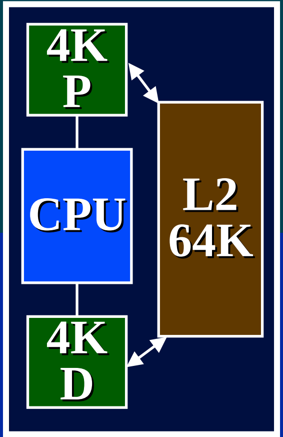
- Async (SRAM, ROM, etc.)
- Sync (SBSRAM, SDRAM)

Internal Memory

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

Level 1 Cache

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


4K P
CPU
4K D
L2 64K