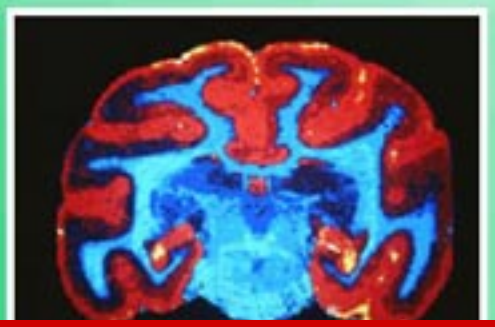
Today's Agenda

- ✔ What are my system requirements?
- ✔ How do I work with TI's 'C6000?
- How do I work with TI's 'C5000?
- How do TI's tools make my development easier?
- What support can I count on?



TMS320C6000

How do I work with TI's 'C6000?

What performance can I expect?

- How do I get my performance?
- How do I interface easily?
- What are the new 'C6000 devices?
- What is my power consumption?
- How does TI enable maximum performance at lower cost?

'C6000: Raw performance

1600 MIPS is a lot of MIPS.

But...

400 MMACS is a lot of processing.

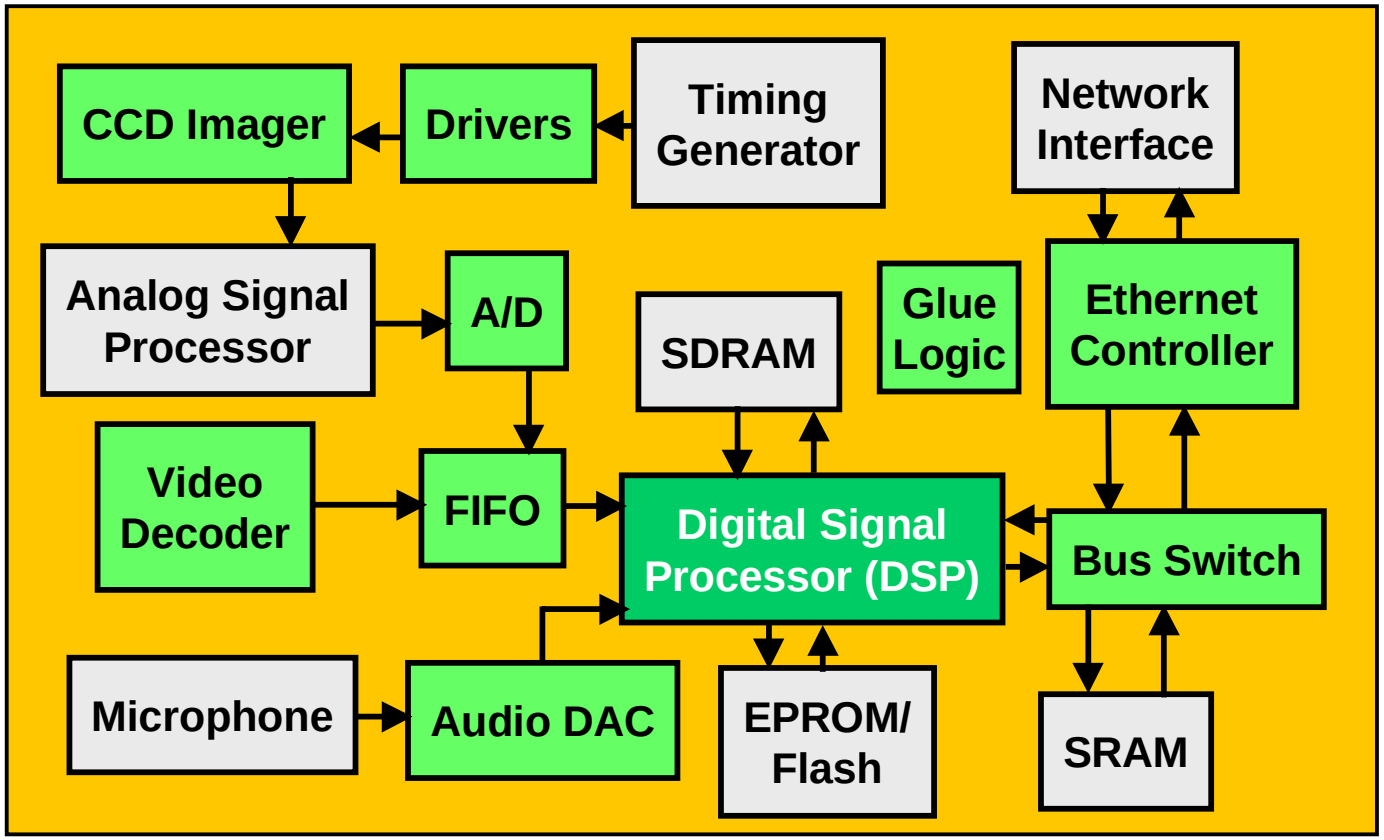
Well, you're right... but...

But what matters to you is how fast you can process *your* algorithm, right?

Show me ...

'C6000: Maximum raw performance

R&D: High-end network security camera system



'C6000: Maximizing multi-function

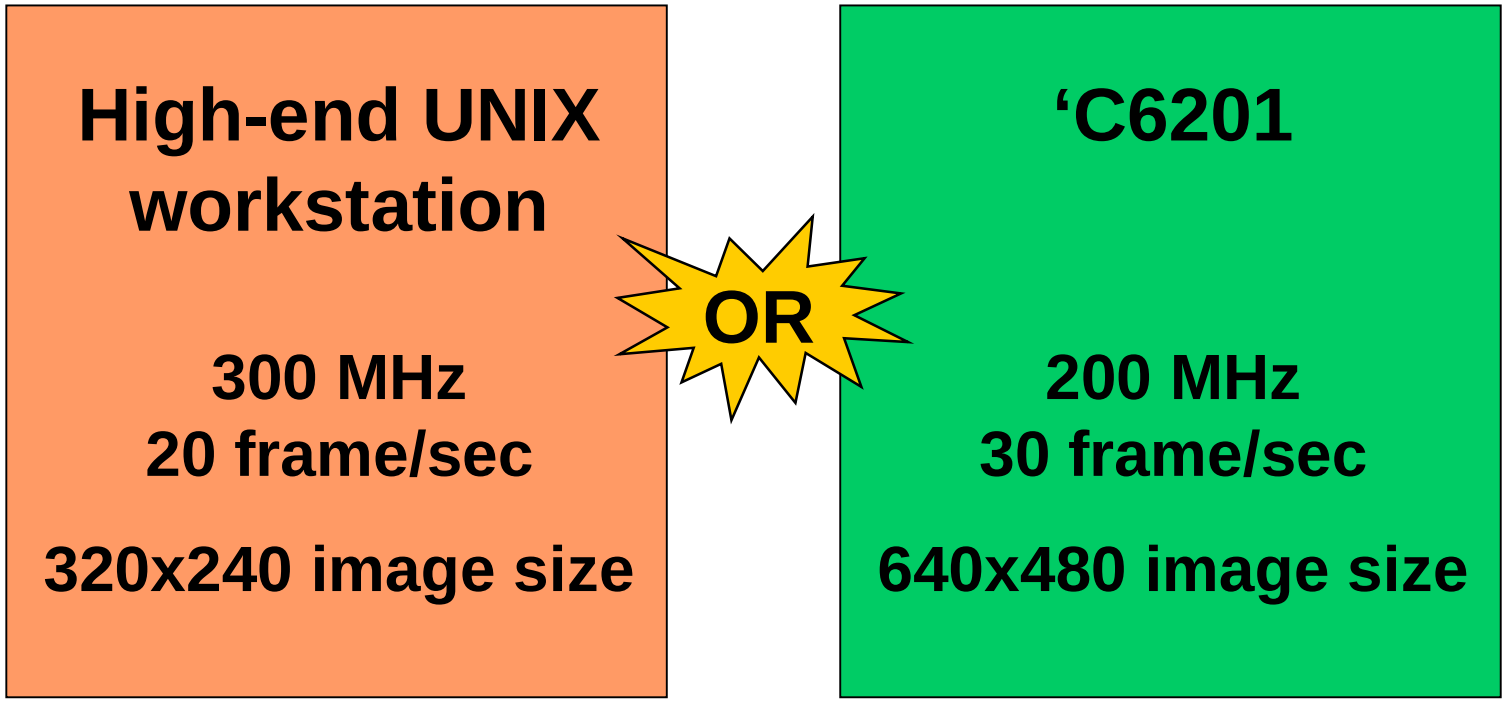
Imaging: Processing-intensive tasks

Algorithms required:

Absolute difference threshold
Morph dilation erosion
Run length encoding
Logical filtering
JPEG compression at 10Hz
Communications/tracking

>30 million instructions/frame

'C6201: Power of a workstation



... with 600 MIPS to spare

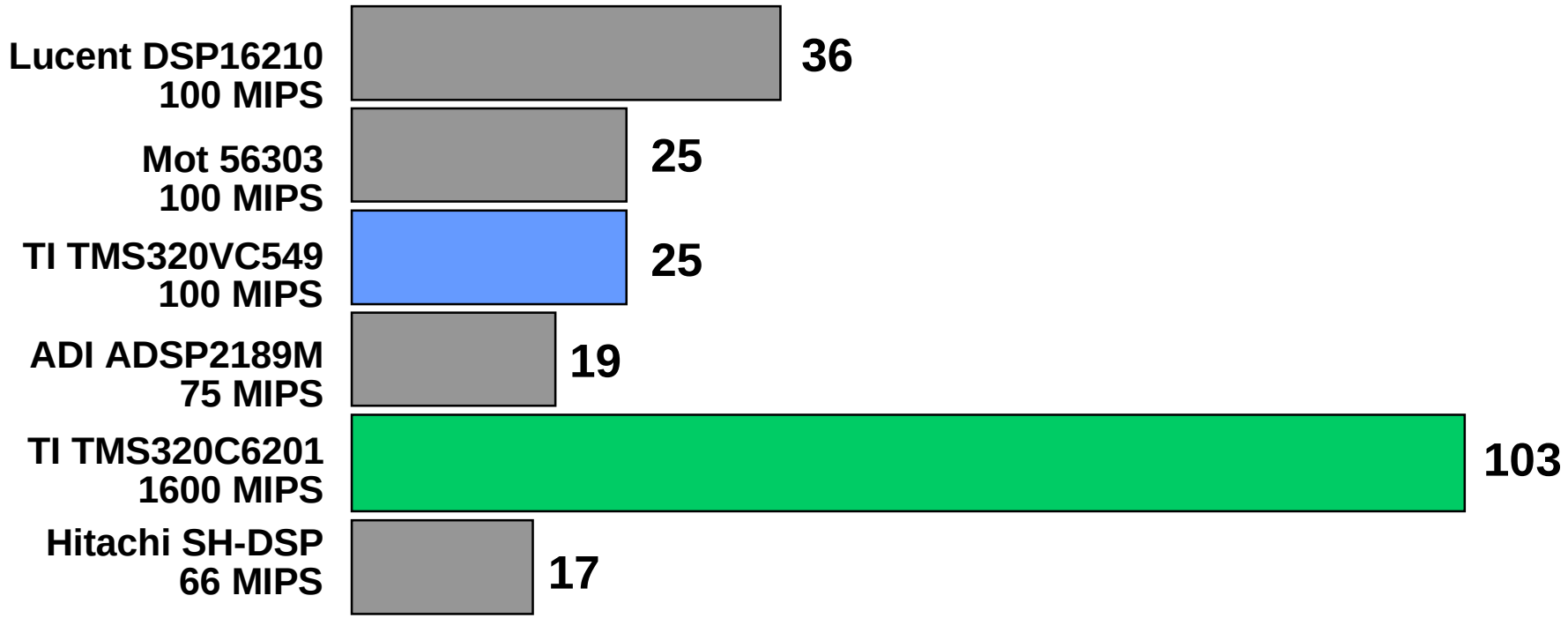
'C6000: Maximizing multi-channel

'C6201	Channels @200 MHz
G.729A	30+
G.723.1	22+
G.726	50-80
Echo Cancellation (32-msec tail)	51
GSM EFR	16+
EVRC	12
V.34/V.90	12-15

'C62x: Raw performance

BDTImark™: A DSP Speed Metric

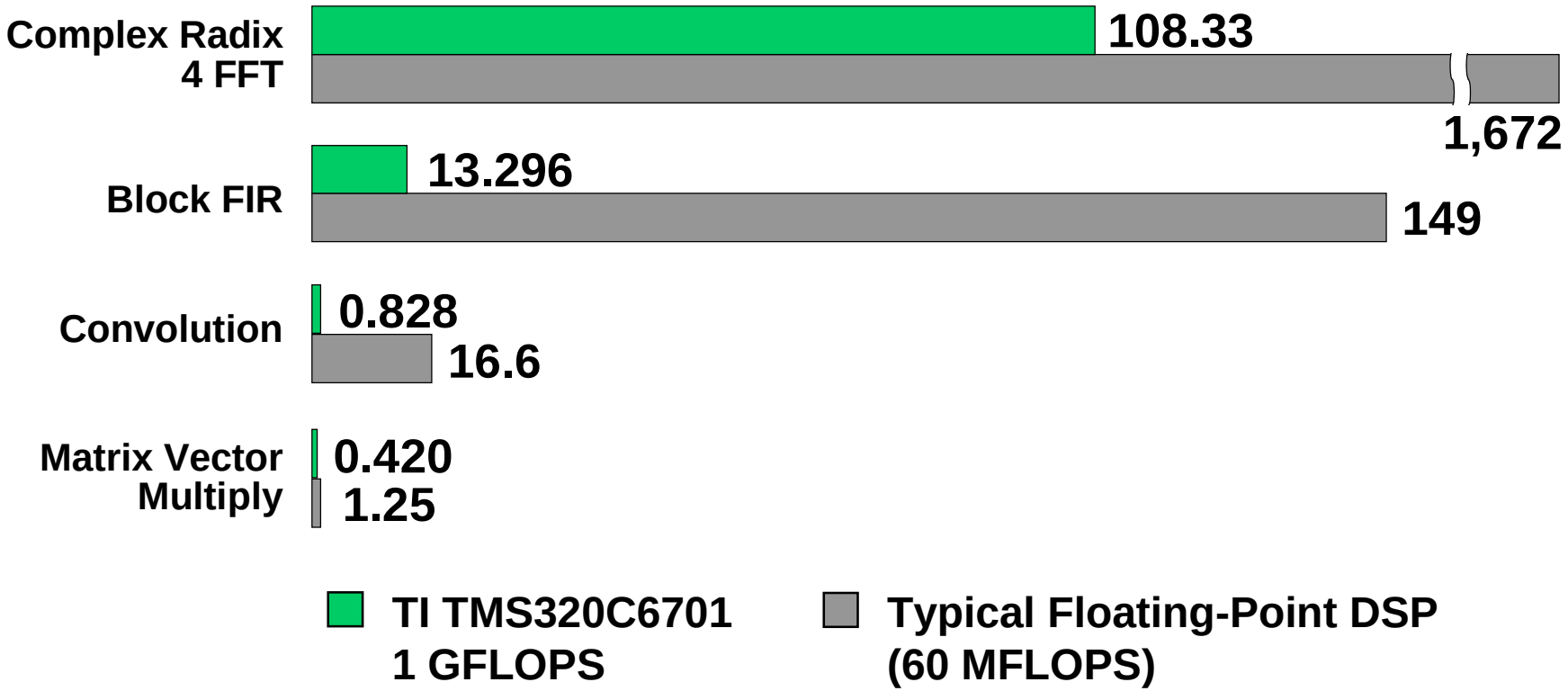
Source www.BDTI.com. ©1999 BDTI



'C67x: Raw performance

Floating-Point Performance

Execution time (in µSec)

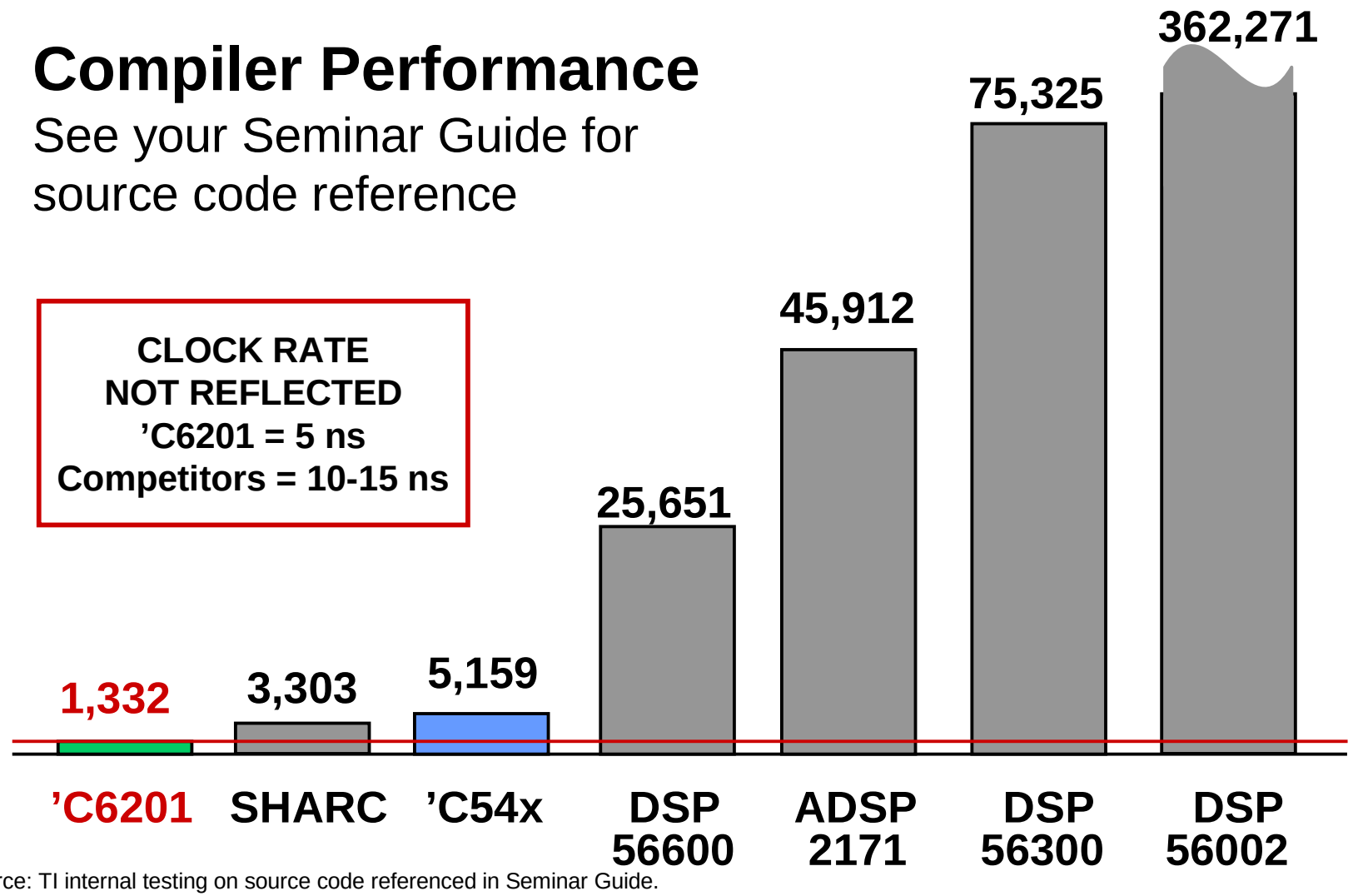


’C6000 Compiler: Lowest cycle count

Compiler Performance

See your Seminar Guide for source code reference

CLOCK RATE NOT REFLECTED
’C6201 = 5 ns
Competitors = 10-15 ns

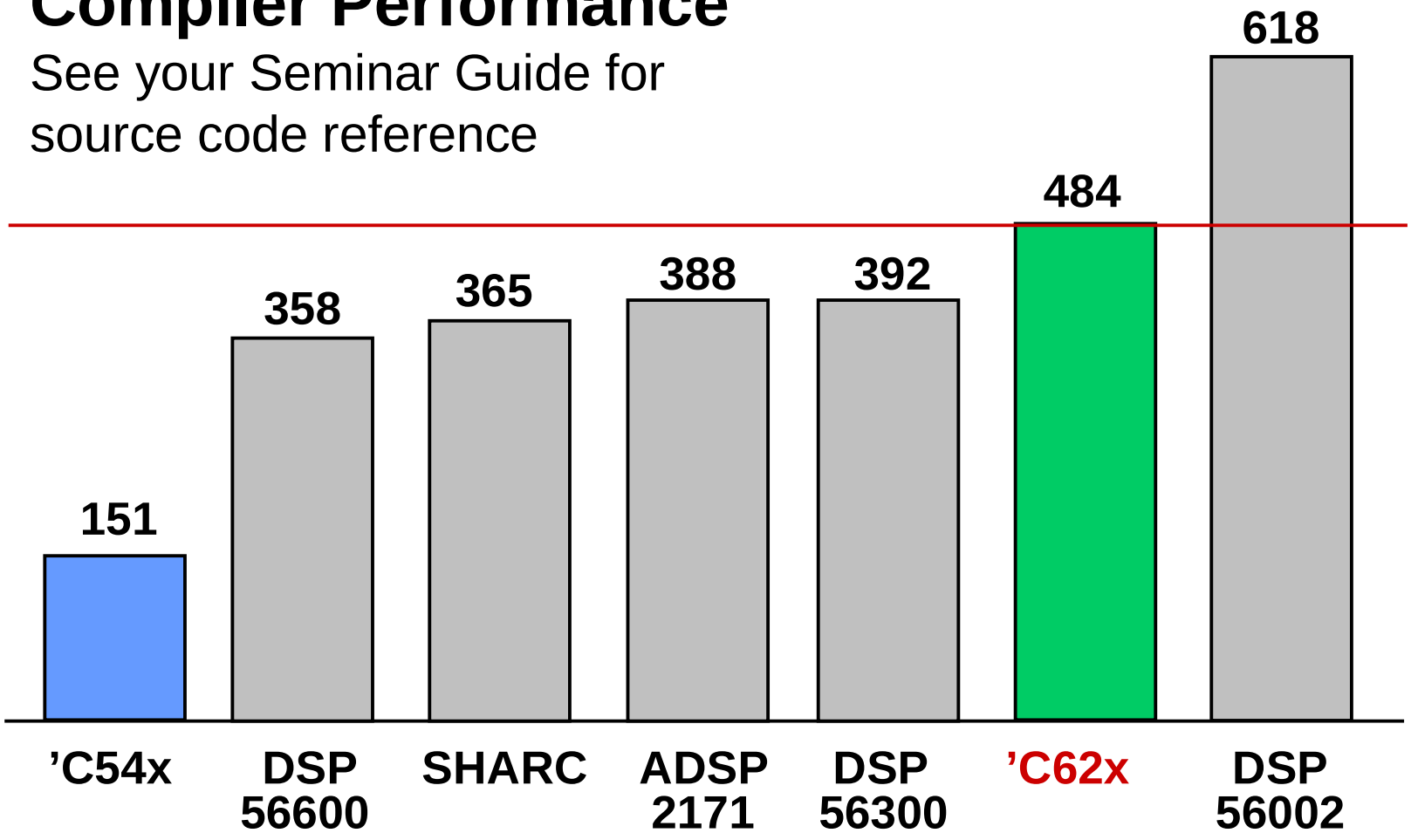


Source: TI internal testing on source code referenced in Seminar Guide.

’C6000 Compiler: Average byte count

Compiler Performance

See your Seminar Guide for source code reference



Source: TI internal testing on source code referenced in Seminar Guide.

How do I work with TI's 'C6000?

What performance can I expect?

How do I get my performance?

How do I interface easily?

What are the new 'C6000 devices?

What is my power consumption?

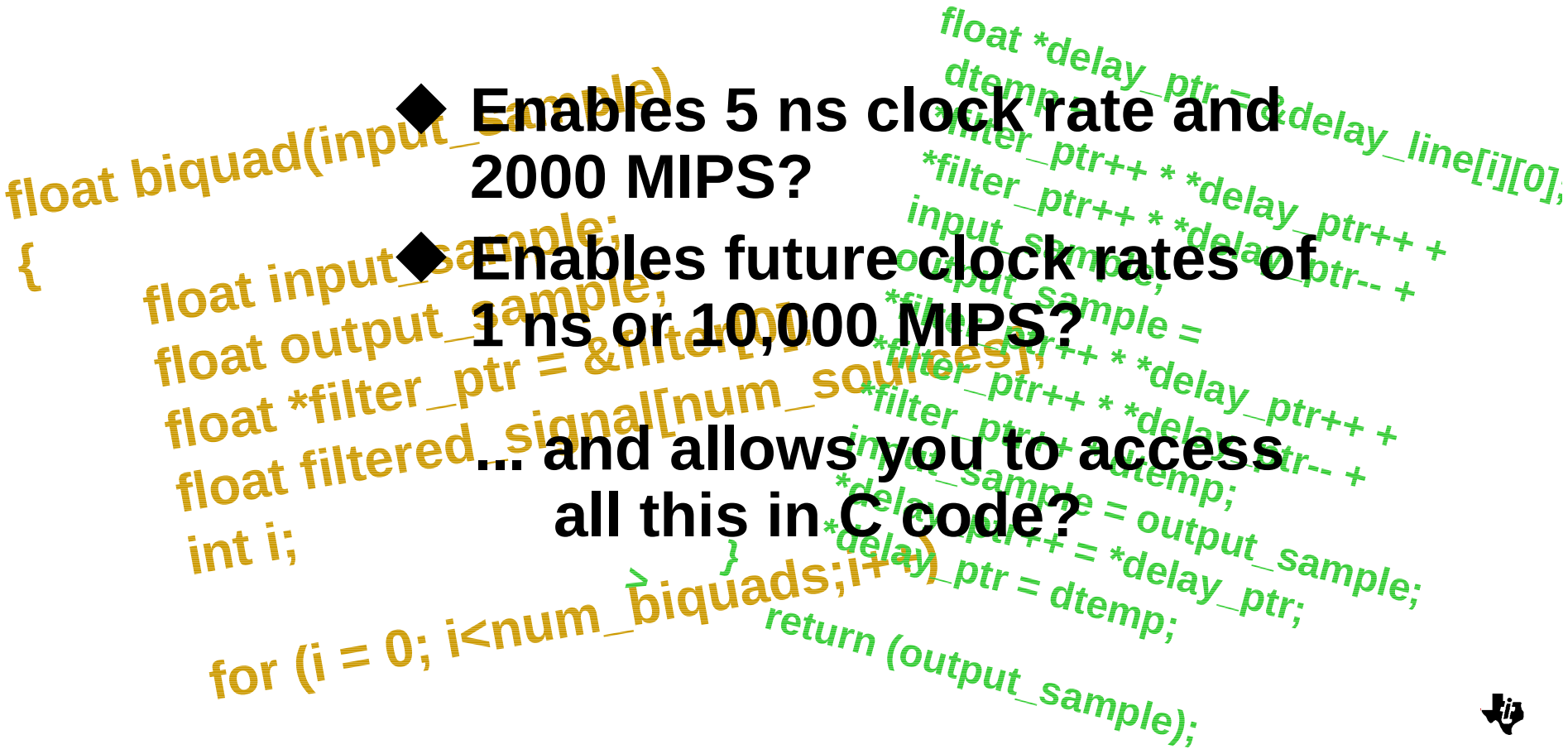
How does TI enable maximum performance at lower cost?

'C6000: Built for speed

What kind of architecture...

- **Enables 5 ns clock rate and 2000 MIPS?**
- **Enables future clock rates of 1 ns or 10,000 MIPS?**

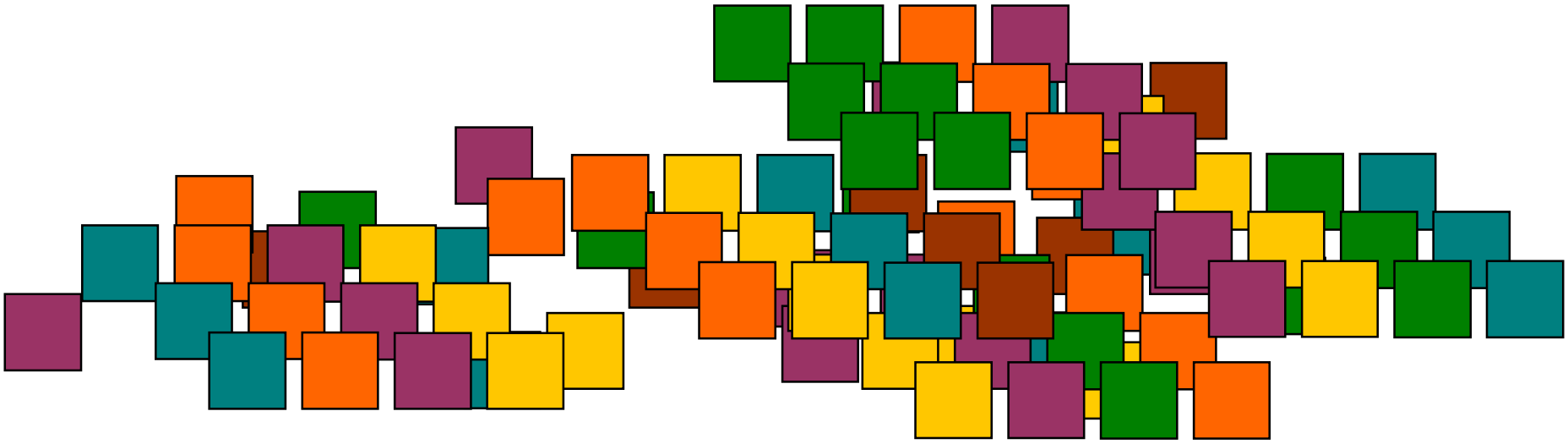
... and allows you to access all this in C code?



Concept: Built for speed

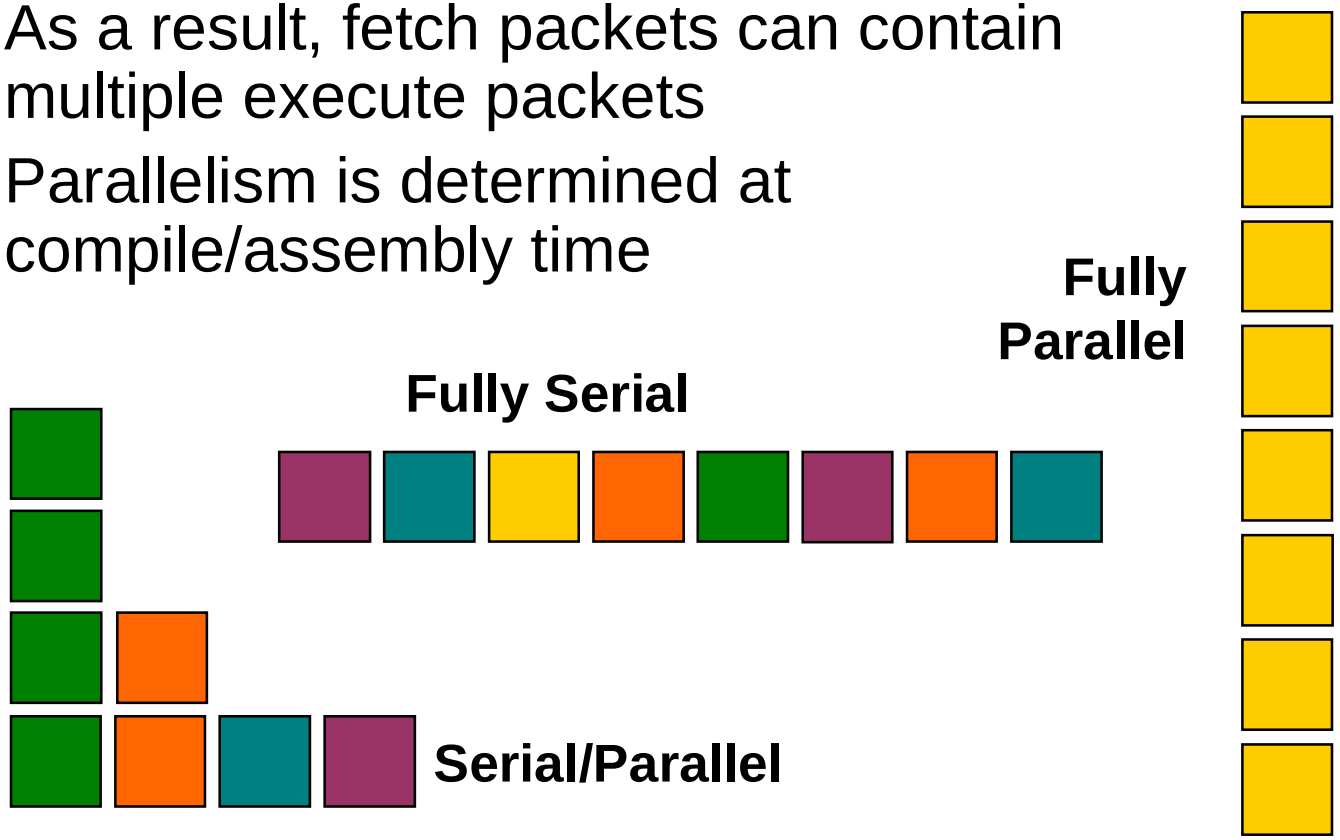
An environment that enables speed might have:

- Lots of blocks that can be moved easily
- Few restrictions on what order they are placed
- A machine designed to move them quickly
- Ability to do as much as possible, simultaneously



VelociTI™: Speed with efficiency

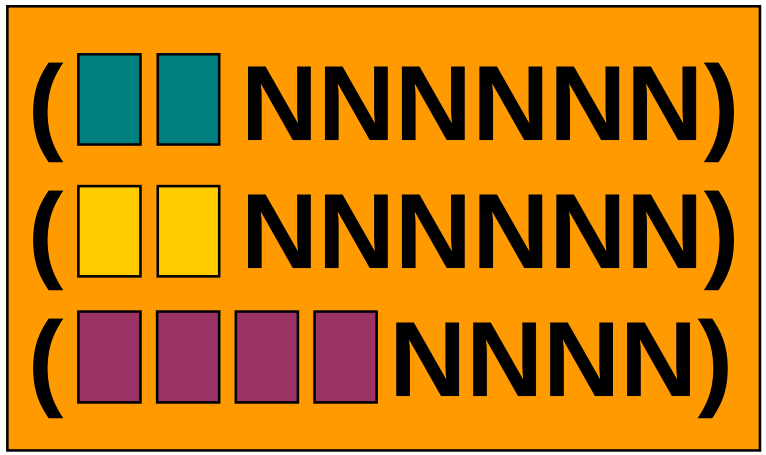
- **Execute:** CPU executes 1 to 8 instructions/cycle
- As a result, fetch packets can contain multiple execute packets
- Parallelism is determined at compile/assembly time



VelociTI™: Advanced VLIW

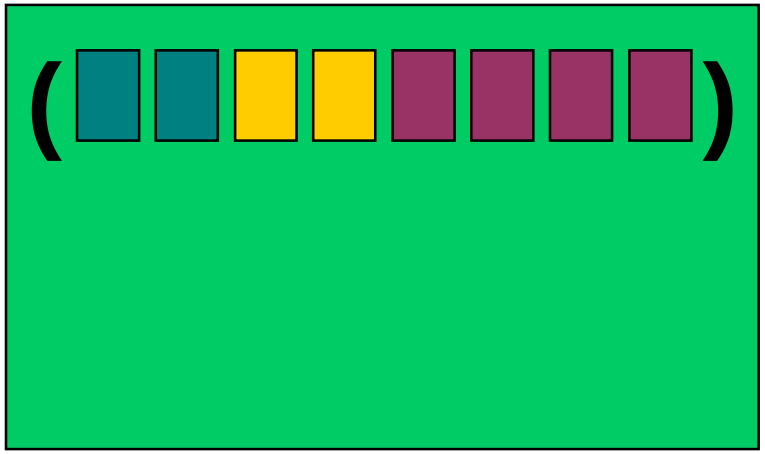
Fetch: CPU fetches 8 instructions/cycle (256 bits)

Traditional VLIW:



8 instructions x 32 bits
...plus NOPs

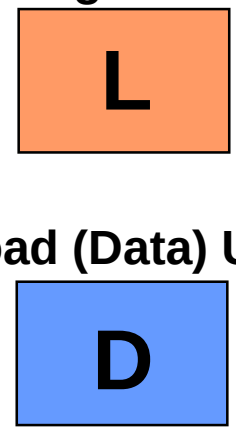
VelociTI:



8 instructions x 32 bits
...with minimal NOPs

'C6000: Shared core

	'C62x Fixed Point	'C67x Floating Point
Shifter Unit	32-bit ALU shifts bit field operations branches	float comparison float ABS reciprocal sq. root reciprocal sngl/dbl prec. conv.
Multiply Unit	16x16 integer multiply	32x32 integer/floating mpy 64 x64 floating mpy
Logic Unit	40-bit ALU integer comparison normalization leftmost bit detect	fix/float conversion float arithmetic
Load (Data) Unit	loads and stores 32-bit add/sub address calculations	doubleword loads



'C6000: Sum of Products example

How is 'C6000 designed to handle math-intensive calculations?

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

Let's look at the two basic instructions required by a MAC -- DSP's fundamental algorithm...

- Multiply
- Add

SOP Example: Add

Where are the variables stored?

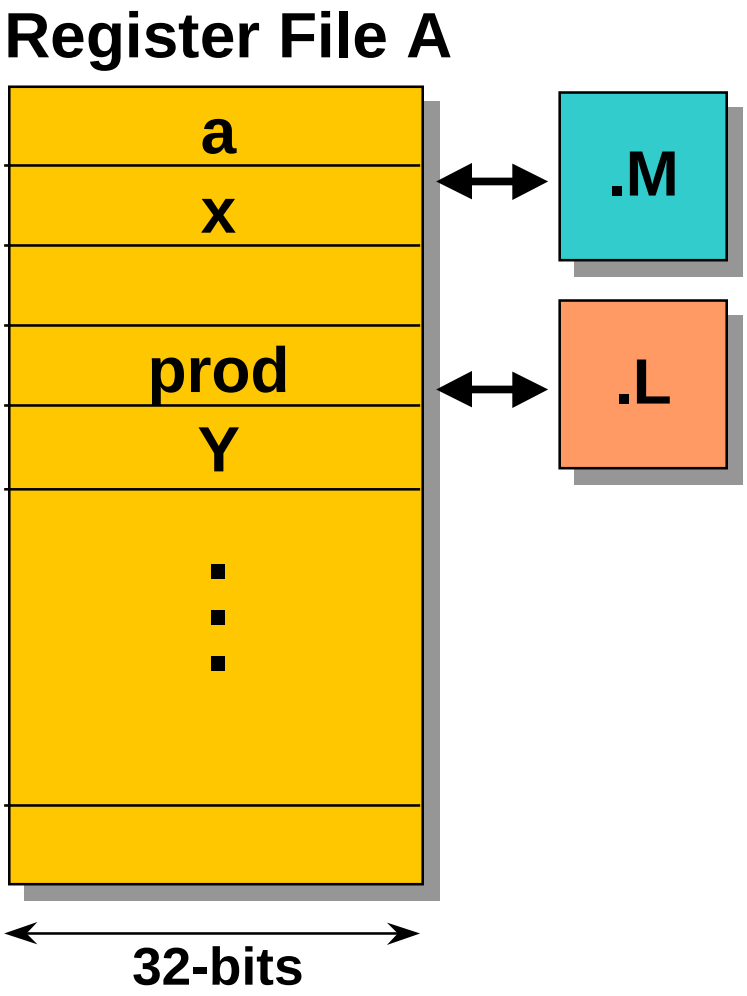
.M

.L

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

MPY	a, x, prod
ADD	Y, prod, Y

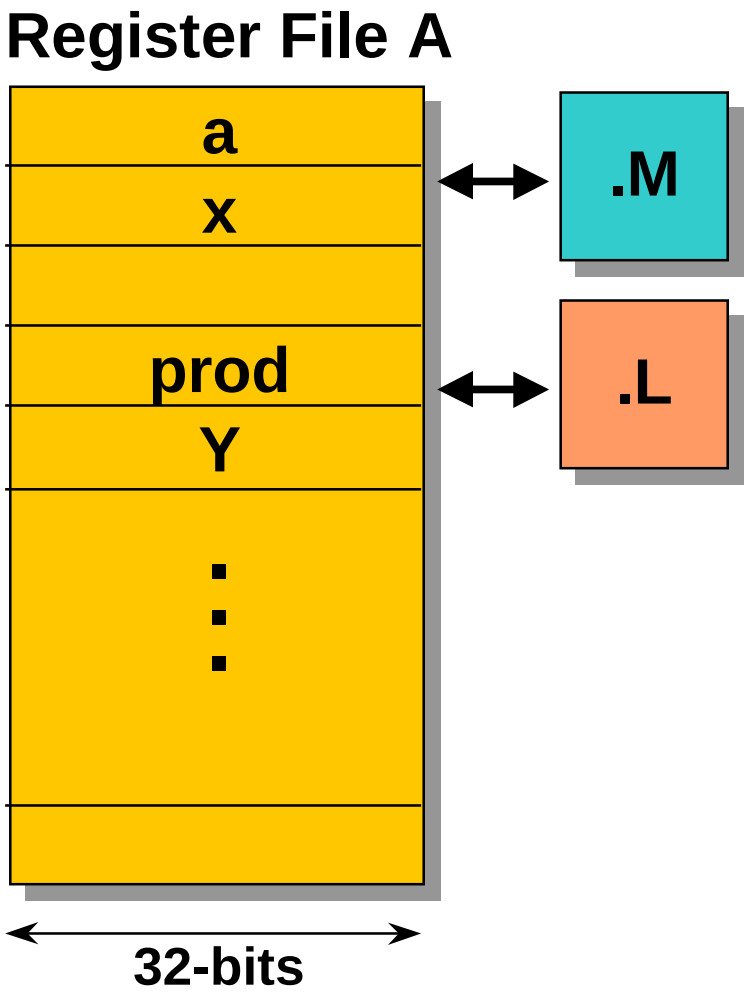
SOP Example: Add



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

```
MPY     a, x, prod
ADD     Y, prod, Y
```

SOP Example: Next, a loop



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

MPY	a, x, prod
ADD	Y, prod, Y

Now, we need a loop

SOP Example: Creating a loop

1. Add branch instruction (B) and a label

```
B       loop
```

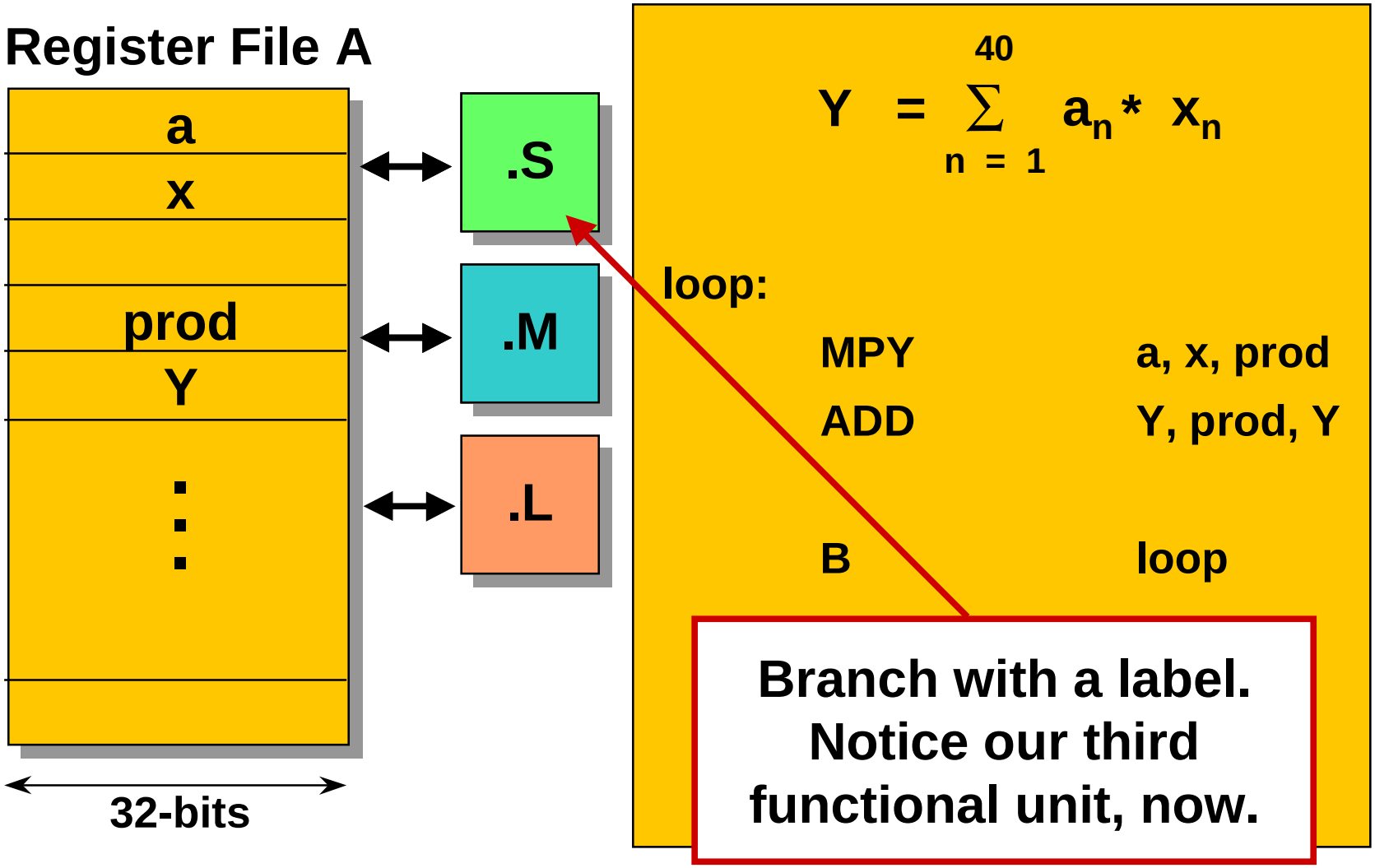
2. Create a loop counter (= 40) counter

```
MVK     40, ctr
```

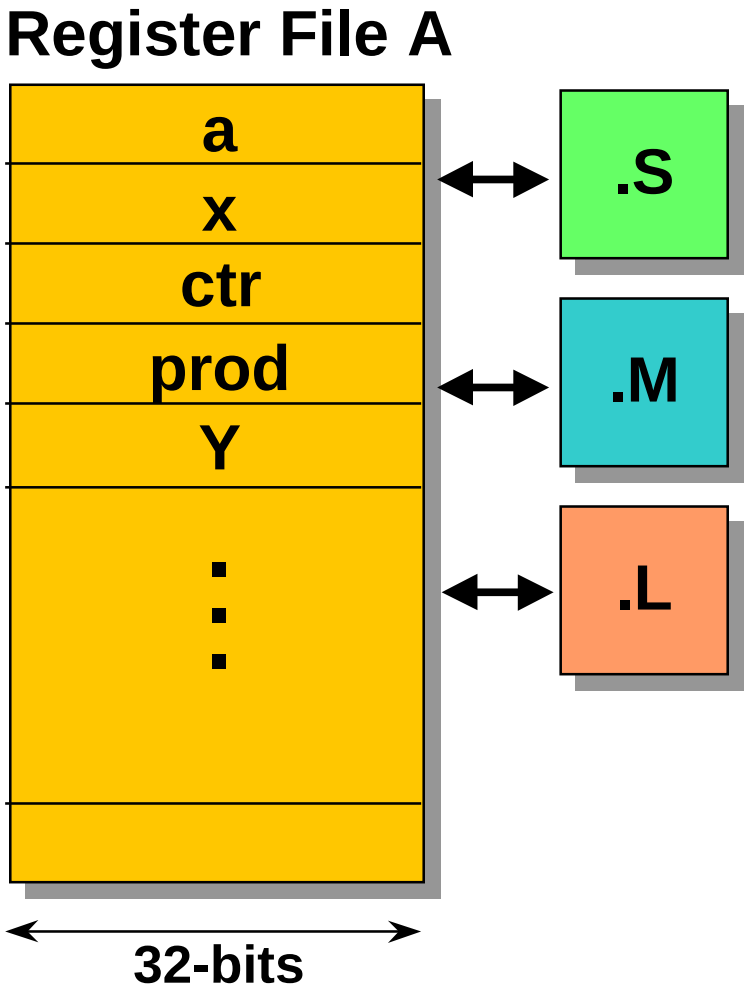
3. Add an instruction to decrement the loop counter

```
SUB     ctr, 1, ctr
```

SOP Example: Decrement counter



SOP Example: Decrement counter

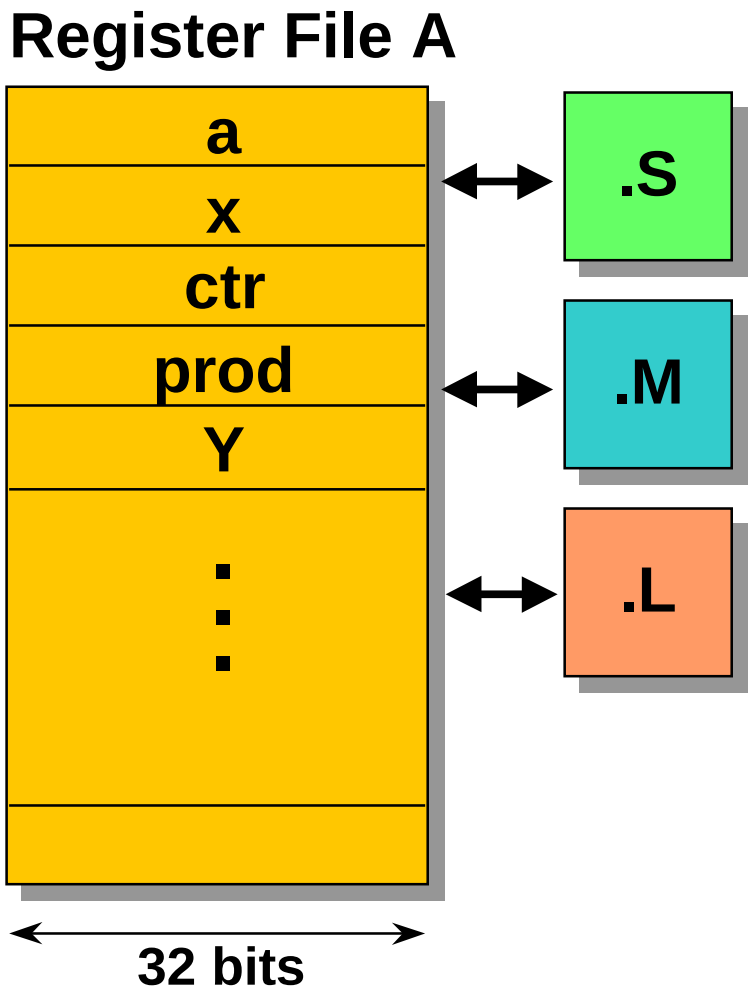


$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
        MVK     40, ctr
loop:
        MPY     a, x, prod
        ADD     Y, prod, Y
        SUB     ctr, 1, ctr
        B       loop
```

This is our decrement counter

SOP Example: Decrement counter



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

```
        MVK     40, ctr
loop:
        MPY     a, x, prod
        ADD     Y, prod, Y
        SUB     ctr, 1, ctr
[ctr]   B       loop
```

Since decrementing is handled by s/w, we use conditional instructions

'C6000: 100% conditional instructions

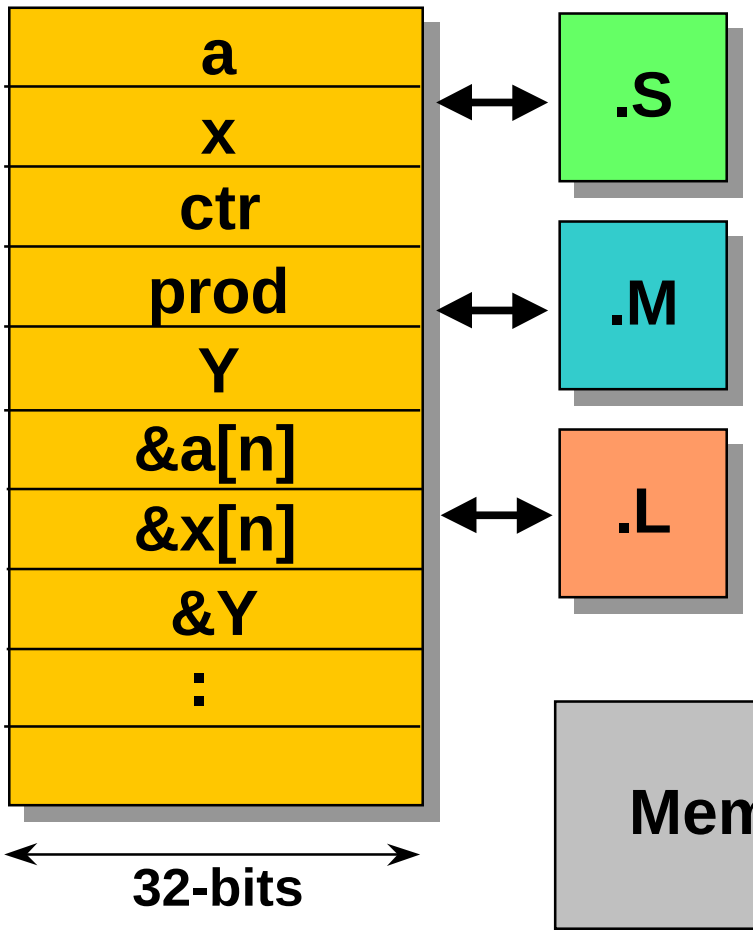
```
[condition]     B          loop
```

In order to reduce branching, all instructions are conditional
based on zero or nonzero values of condition variables.

Code Syntax	Execute instruction if :
[ctr]	**ctr ≠ 0**
[!ctr]	**ctr = 0**

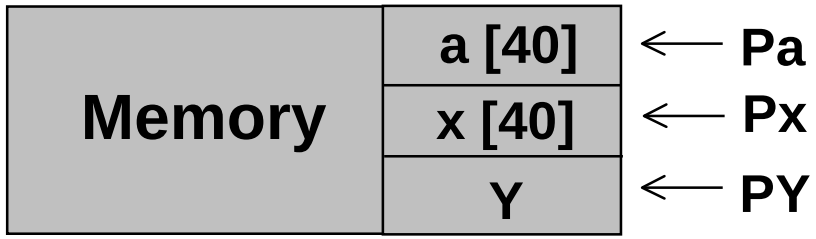
Note: if condition is false, NOP is executed

SOP Example: Loading values

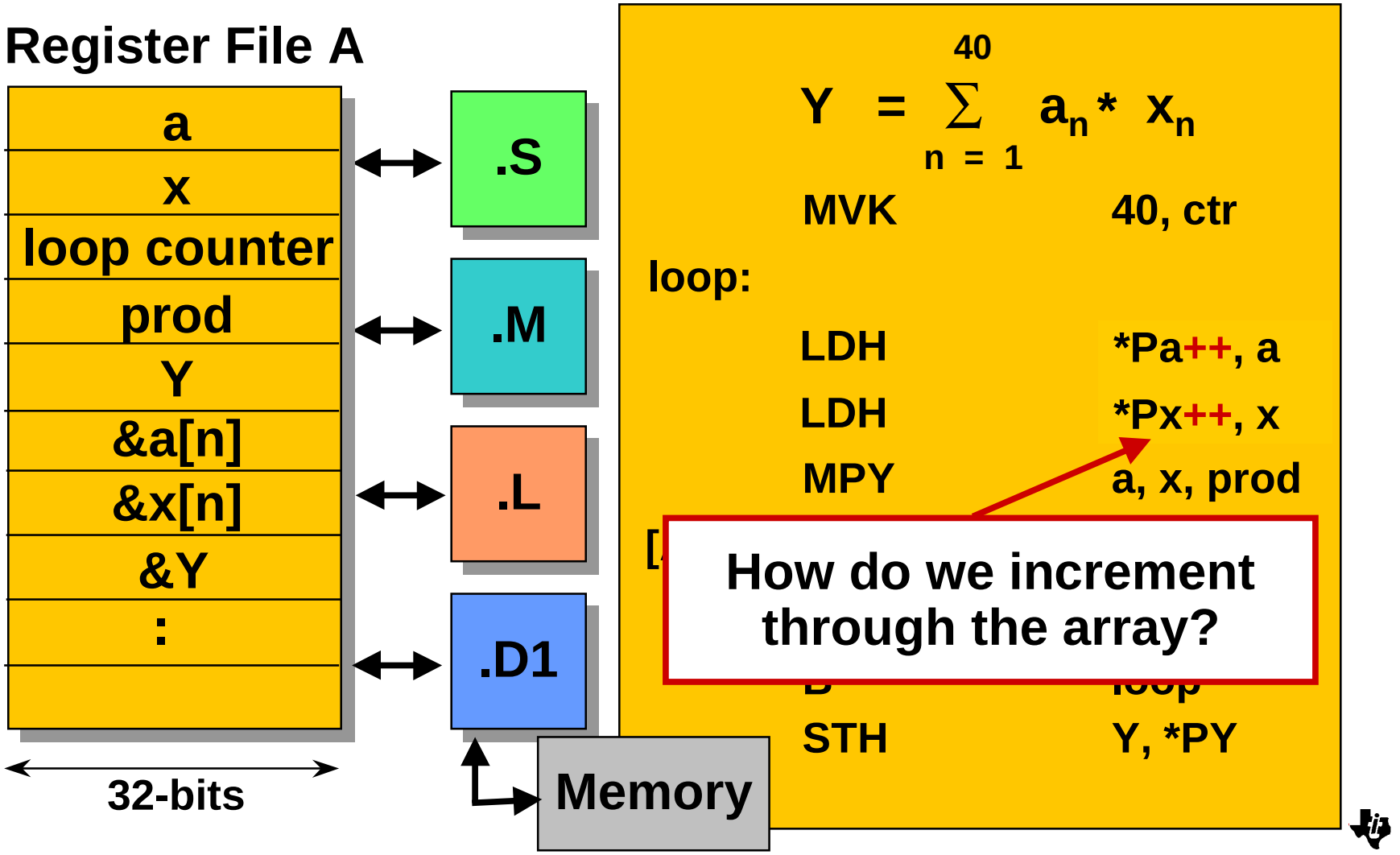


How are a and x loaded?

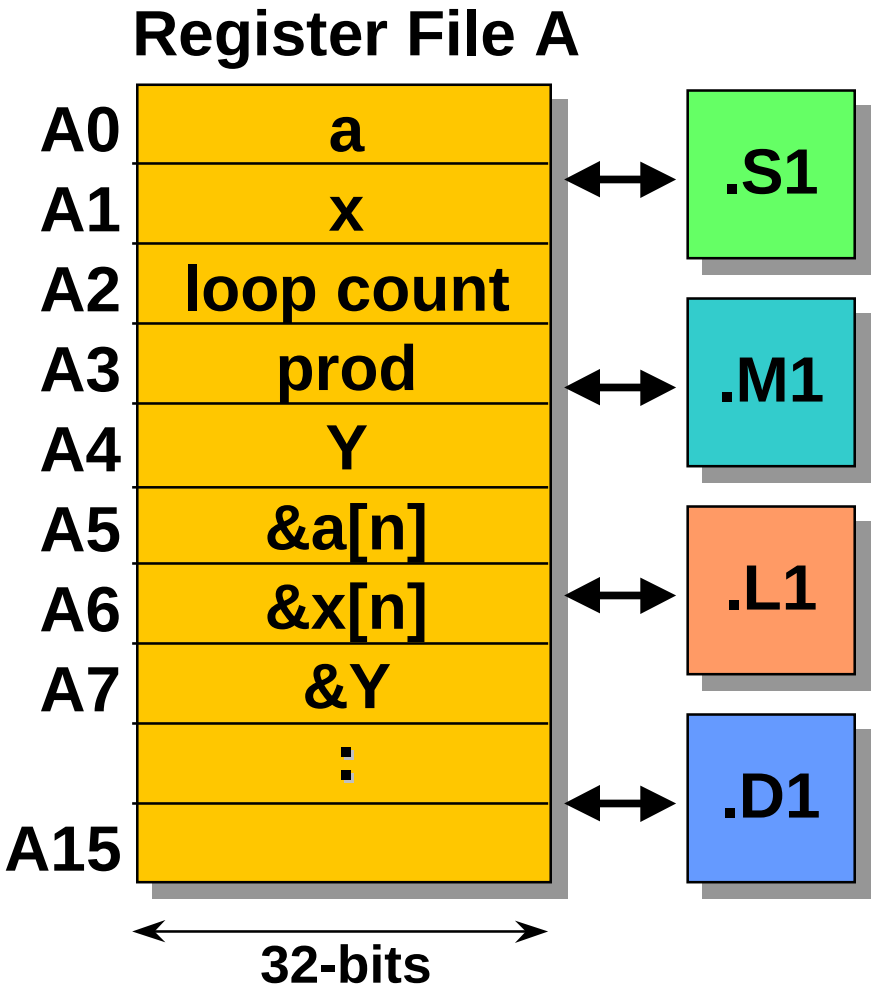
- a, x, Y located in memory
- Create a pointer to values
  - Pa = &a
  - Px = &x
  - PY = &Y



SOP Example: Loading values



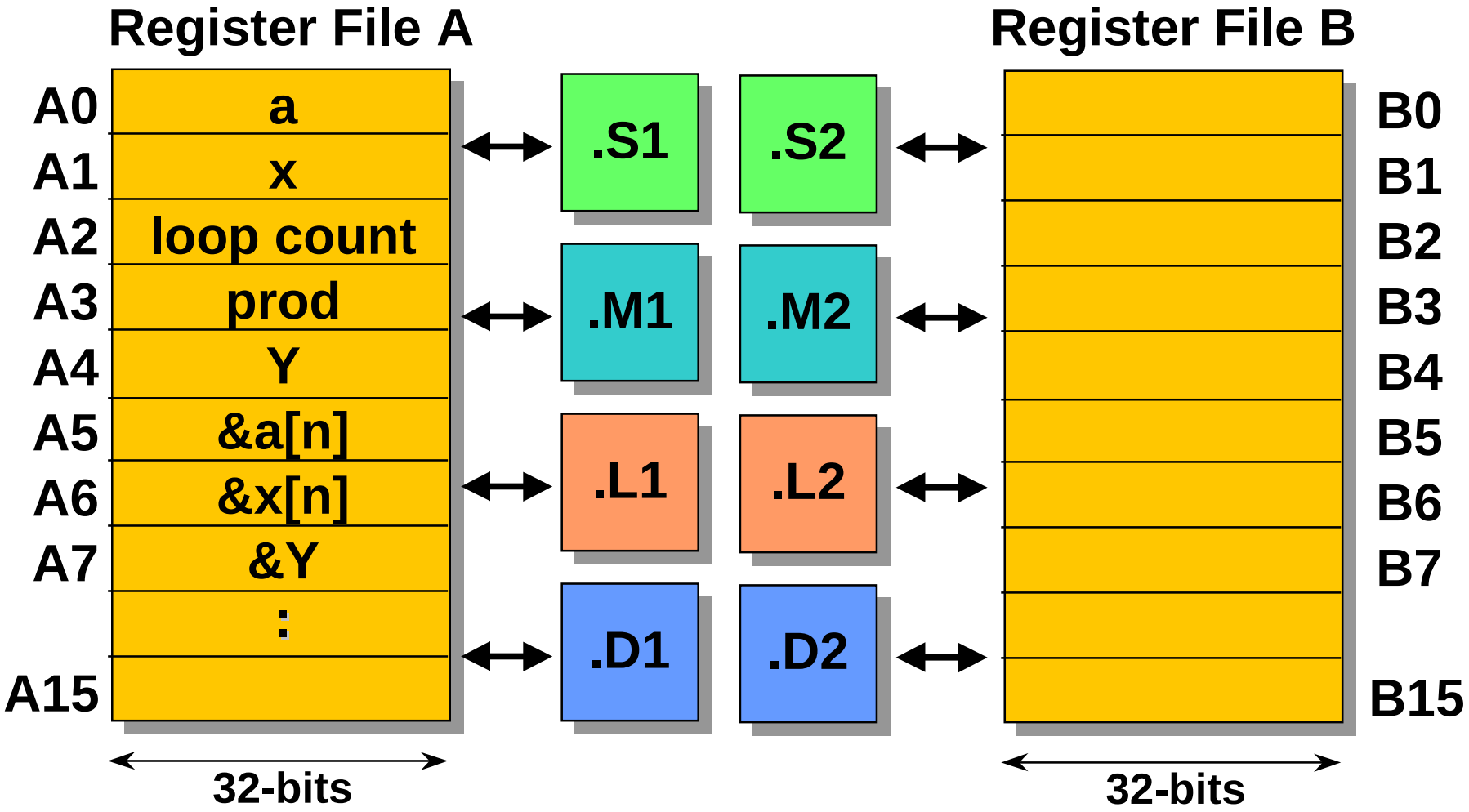
'C62x ASM: Sum of Products



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

```
        MVK   .S1   40, A2
loop:   LDH   .D1   *A5++, A0
        LDH   .D1   *A6++, A1
        MPY   .M1   A0, A1, A3
        ADD   .L1   A4, A3, A4
        SUB   .S1   A2, 1, A2
[A2]    B     .S1   loop
        STH   .D1   A4, *A7
```

'C62x ASM: Dual paths



'C62x: RISC-like instruction set

.S Unit		
ADD	NEG	
ADDK	NOT	
ADD2	OR	
AND	SET	
B	SHL	
CLR	SHR	
EXT	SSHL	
MV	**SUB**	
MVC	SUB2	
MVK	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	ST
ADDA	**SUB**
LD	SUBA
MV	ZERO
NEG	

No Unit Used	
NOP	IDLE

'C67x: Superset of fixed-point set

.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*
MVKH	ZERO	*RSQRSP*
		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

.M Unit		
MPY	SMPY	*MPYSP*
MPYH	SMPYH	*MPYDP*
		MPYI
		MPYID

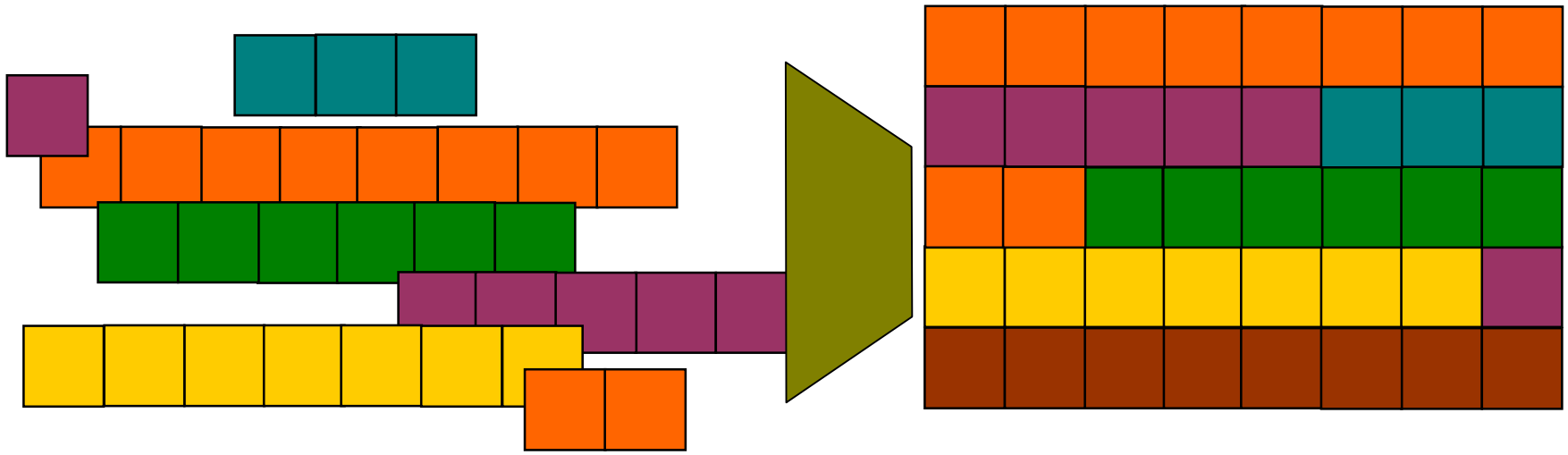
.D Unit	
ADD	ST
ADDA	**SUB**
LD	SUBA
MV	ZERO
NEG	

No Unit Used	
NOP	IDLE

VelociTI: A true C engine

VelociTI is designed as an ideal target for the ’C6000 C compiler, enabling highly efficient C code development...

... with the compiler handling scheduling for you



'C62x: Generic C code

Sum of Products

```
#define  N          40

         short      a[N] = {0x0001, 0x0002, 0x0003, 0x0004,  . . .   };
         short      x[N] = {0x0005, 0x0006, 0x0007, 0x0008,   . . .   };

int sumprod(void) {
         char       i;
         int        sum=0;

         for(i=0; i<N; i++) {
                    sum += a[i] * x[i];
         }

         return(sum);
}
```

Optimized output = 2 taps per cycle

Two Sums of Product per iteration

```
L2:     ; PIPED LOOP PROLOG
        LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

        LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
```

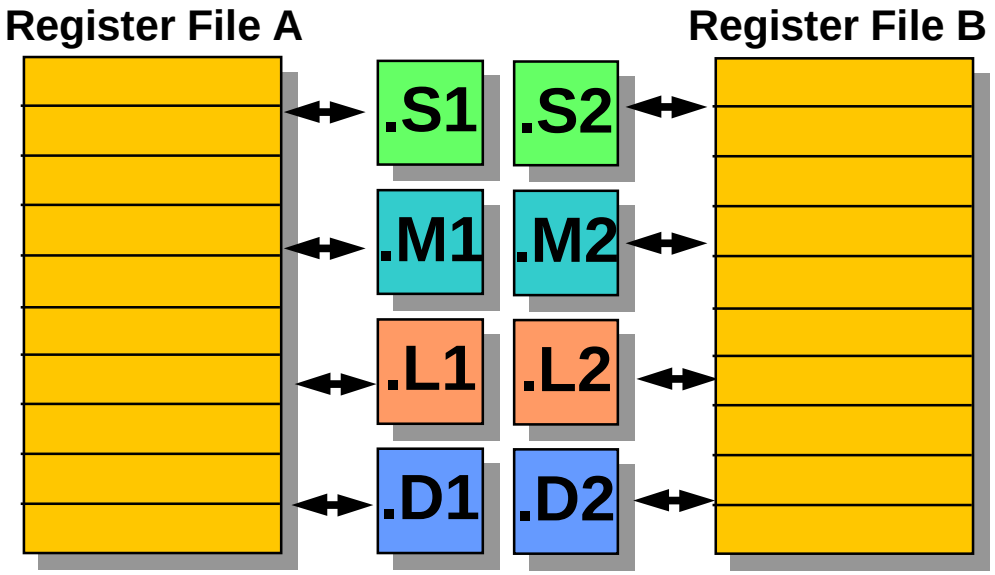
```
        MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

        MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
;** ----------------------------------*
L3:     ; PIPED LOOP KERNEL
        ADD     .L2     B4,B5,B5
||      ADD     .L1     A5,A0,A0
||      MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
|| [ B0] SUB    .S2     B0,1,B0
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
;** ----------------------------------*
```

How can we get 40 taps in 28 cycles?

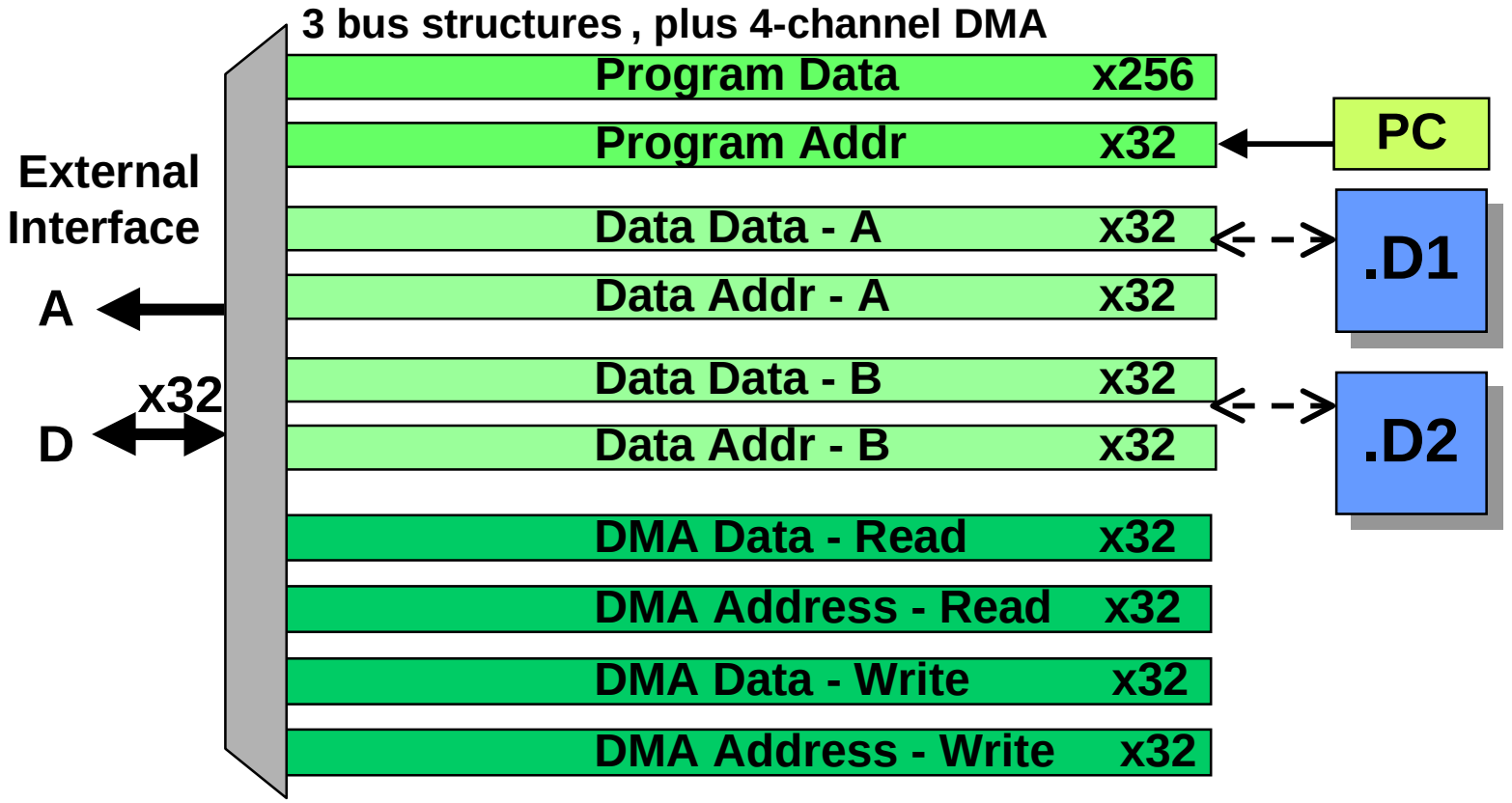
Key #1: Dual paths create two results per loop.

We split the problem in half, and worked on the halves simultaneously... doing intermediate calculations two at a time.



Busses: Supporting 5-ns clock rate

Key #2: Multiple busses allow dual data access and program fetch simultaneously.



Concept: Software pipelining

Key #3: Software pipelining enables efficiency.

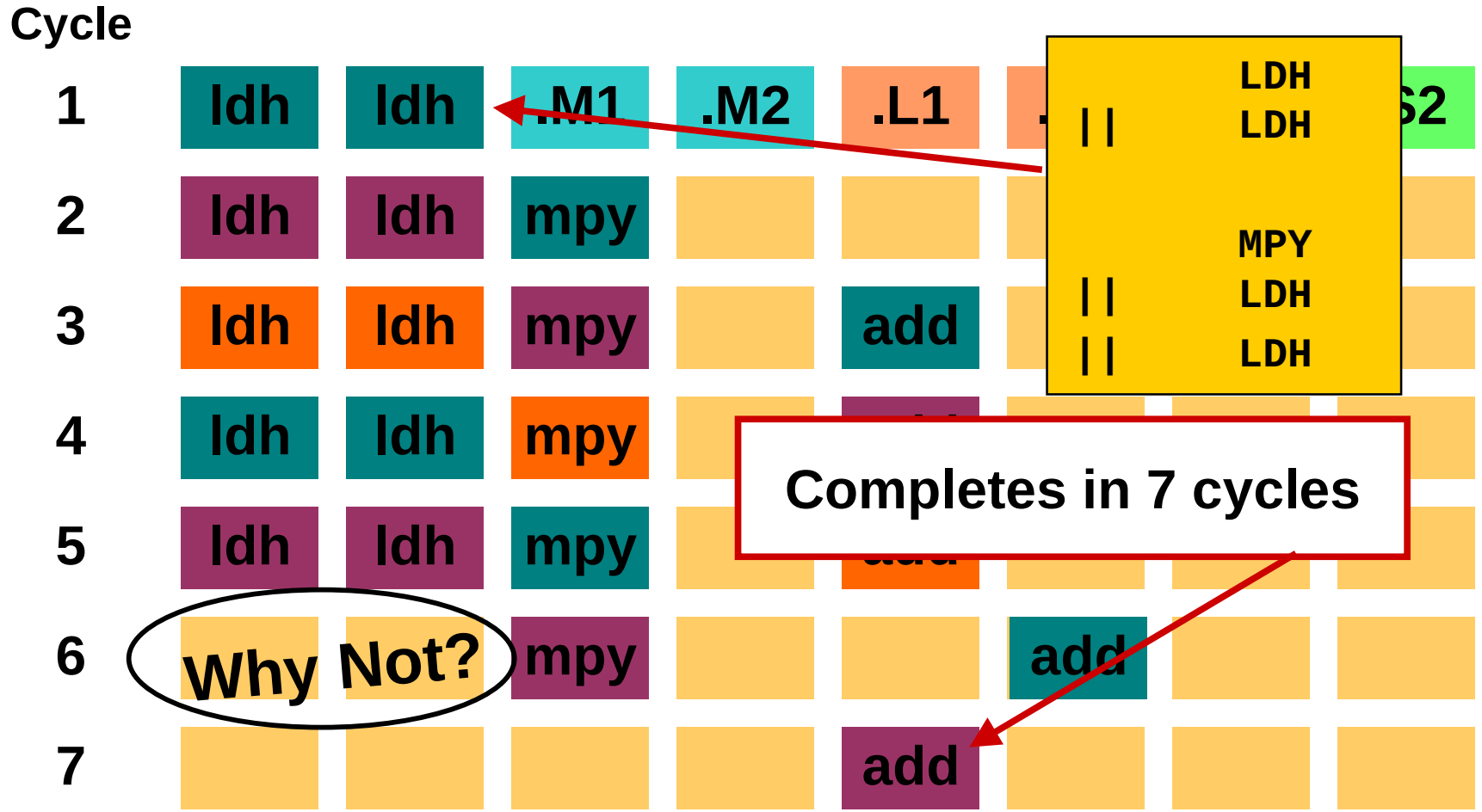
```
     LDH
||   LDH
     MPY
     ADD
```

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

5 x 3 = 15 cycles

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

Software Pipelining: 1/2 the cycles



Results: 2x taps; half the cycles

#1 -- Dual paths enable 2 taps per cycle

```
L2:     ; PIPED LOOP PROLOG
        LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

        LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
```

```
        MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

        MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
;** ---------------------------------*
L3:     ; PIPED LOOP KERNEL
        ADD     .L2     B4,B5,B5
||      ADD     .L1     A5,A0,A0
||      MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
|| [ B0] SUB    .S2     B0,1,B0
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
;** ---------------------------------*
```

Results: 2x taps; half the cycles

#2 -- Busses enable multiple data loads

```
L2:     ; PIPED LOOP PROLOG
        LDW    .D1T1   *A4++,A3
||      LDW    .D2T2   *B6++,B7

        LDW    .D1T1   *A4++,A3
||      LDW    .D2T2   *B6++,B7

 [ B0]  B      .S1     L3
||      LDW    .D1T1   *A4++,A3
||      LDW    .D2T2   *B6++,B7

 [ B0]  B      .S1     L3
||      LDW    .D1T1   *A4++,A3
||      LDW    .D2T2   *B6++,B7

 [ B0]  B      .S1     L3
||      LDW    .D1T1   *A4++,A3
||      LDW    .D2T2   *B6++,B7
```

```
        MPY    .M2X    B7,A3,B4
||      MPYH   .M1X    B7,A3,A5
|| [ B0] B     .S1     L3
||      LDW    .D1T1   *A4++,A3
||      LDW    .D2T2   *B6++,B7

        MPY    .M2X    B7,A3,B4
||      MPYH   .M1X    B7,A3,A5
|| [ B0] B     .S1     L3
||      LDW    .D1T1   *A4++,A3
||      LDW    .D2T2   *B6++,B7
;** ---------------------------------*
L3:     ; PIPED LOOP KERNEL
        ADD    .L2     B4,B5,B5
||      ADD    .L1     A5,A0,A0
||      MPY    .M2X    B7,A3,B4
||      MPYH   .M1X    B7,A3,A5
|| [ B0] B     .S1     L3
|| [ B0] SUB   .S2     B0,1,B0
||      LDW    .D1T1   *A4++,A3
||      LDW    .D2T2   *B6++,B7
;** ---------------------------------*
```

Results: 2x taps; half the cycles

#3 -- Software pipelined = maximum parallelism

```
L2:     ; PIPED LOOP PROLOG
        LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

        LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

 [ B0]  B       .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
```

```
        MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7

        MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
;** ---------------------------------*
L3:     ; PIPED LOOP KERNEL
        ADD     .L2     B4,B5,B5
||      ADD     .L1     A5,A0,A0
||      MPY     .M2X    B7,A3,B4
||      MPYH    .M1X    B7,A3,A5
|| [ B0] B      .S1     L3
|| [ B0] SUB    .S2     B0,1,B0
||      LDW     .D1T1   *A4++,A3
||      LDW     .D2T2   *B6++,B7
;** ---------------------------------*
```

'C6000: Performance summary

- ✔ The true measure of performance is the speed of the architecture on *your* algorithm.
- ✔ 'C6000's VelociTI architecture achieves maximum raw performance.
- ✔ VelociTI's RISC-like design is an ideal target that enables the C compiler to schedule instructions for maximum parallelism.
- ✔ Programming in C guarantees quick, easy development on a powerhouse device.

How do I work with TI's 'C6000?

What performance can I expect?

How do I get my performance?

How do I interface easily?

What are the new 'C6000 devices?

What is my power consumption?

How does TI enable maximum performance at lower cost?

'C6000: Interfacing

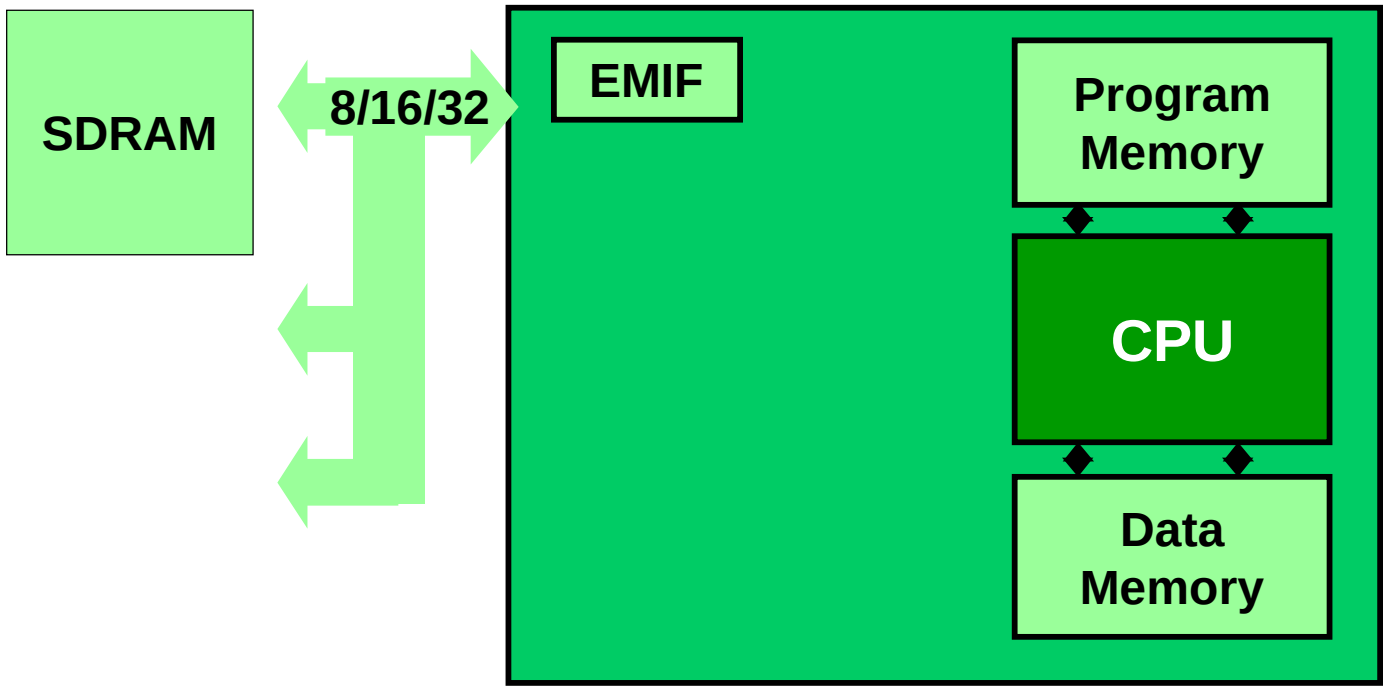
Varied sources for flexible interfacing:

- ✔ Internal Memory
- ✔ External Memory Interface (EMIF)
- ✔ Host Port Interface (HPI)
- ✔ New Expansion Port
- ✔ Multi-channel Buffered Serial Port (McBSP)
- ✔ Direct Memory Access (DMA)

Internal Memory: Feeding the CPU

	L1 Memory		L2 Memory
	Program Memory	**Data Memory**	
'C6201	**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	**4 KB** 1 blk Cache	**4 KB** 1 blk Cache	**64 KB** 4 blk Mapped Cache

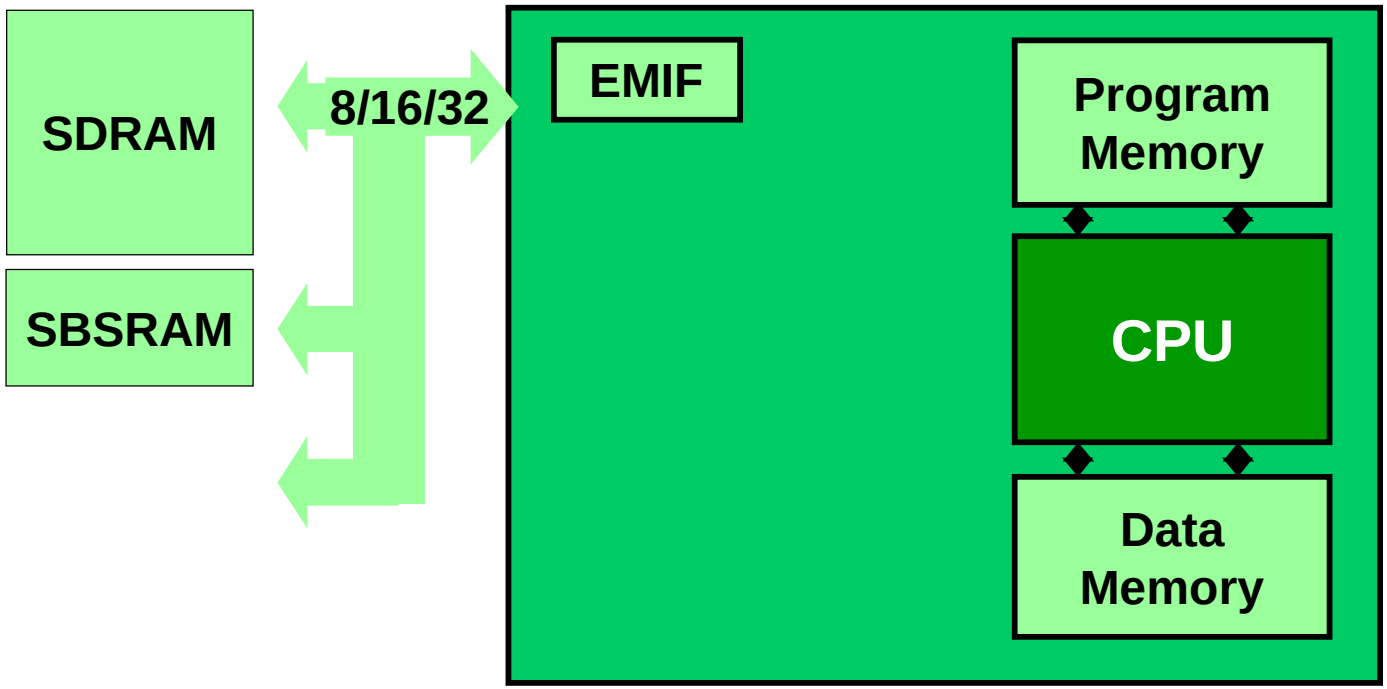
Interfacing: External memory I/F



SDRAM

Provides lowest cost/bit and operates up to 125MHz

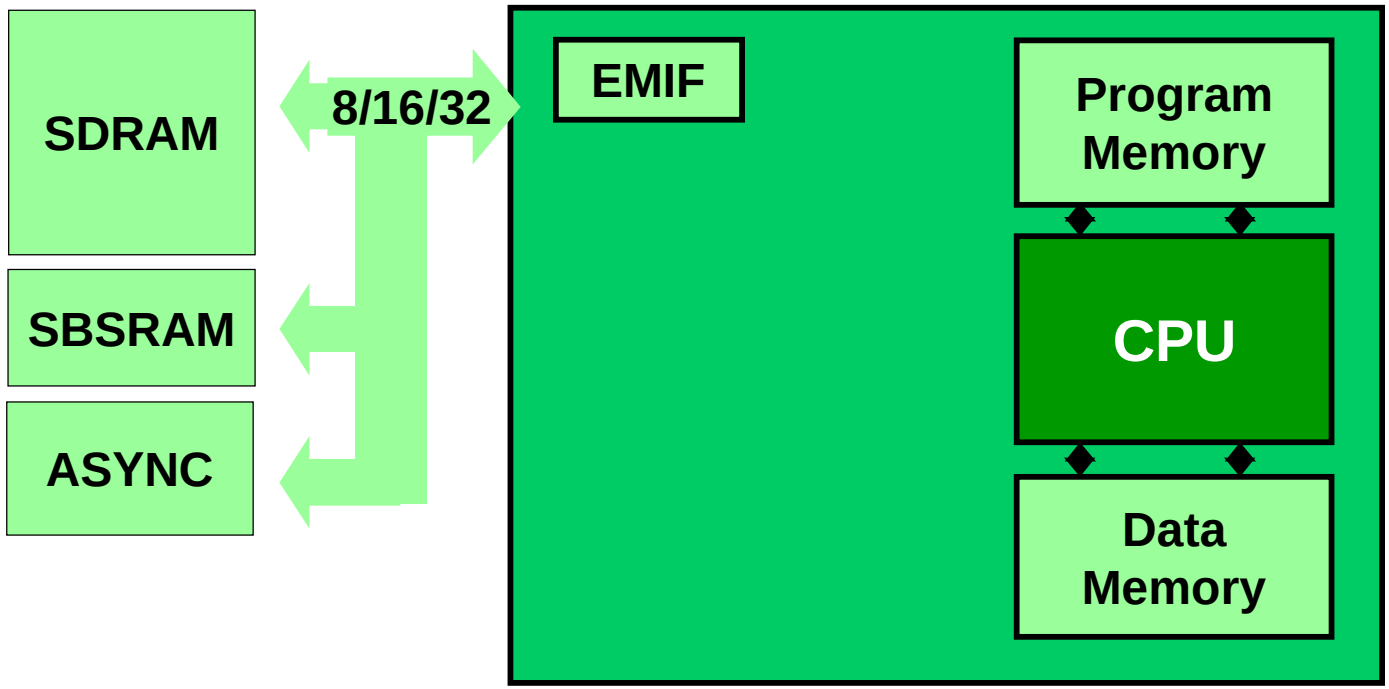
Interfacing: External memory I/F



SBSRAM

I/F allows higher sustained throughput with no refresh/page penalty

Interfacing: External memory I/F



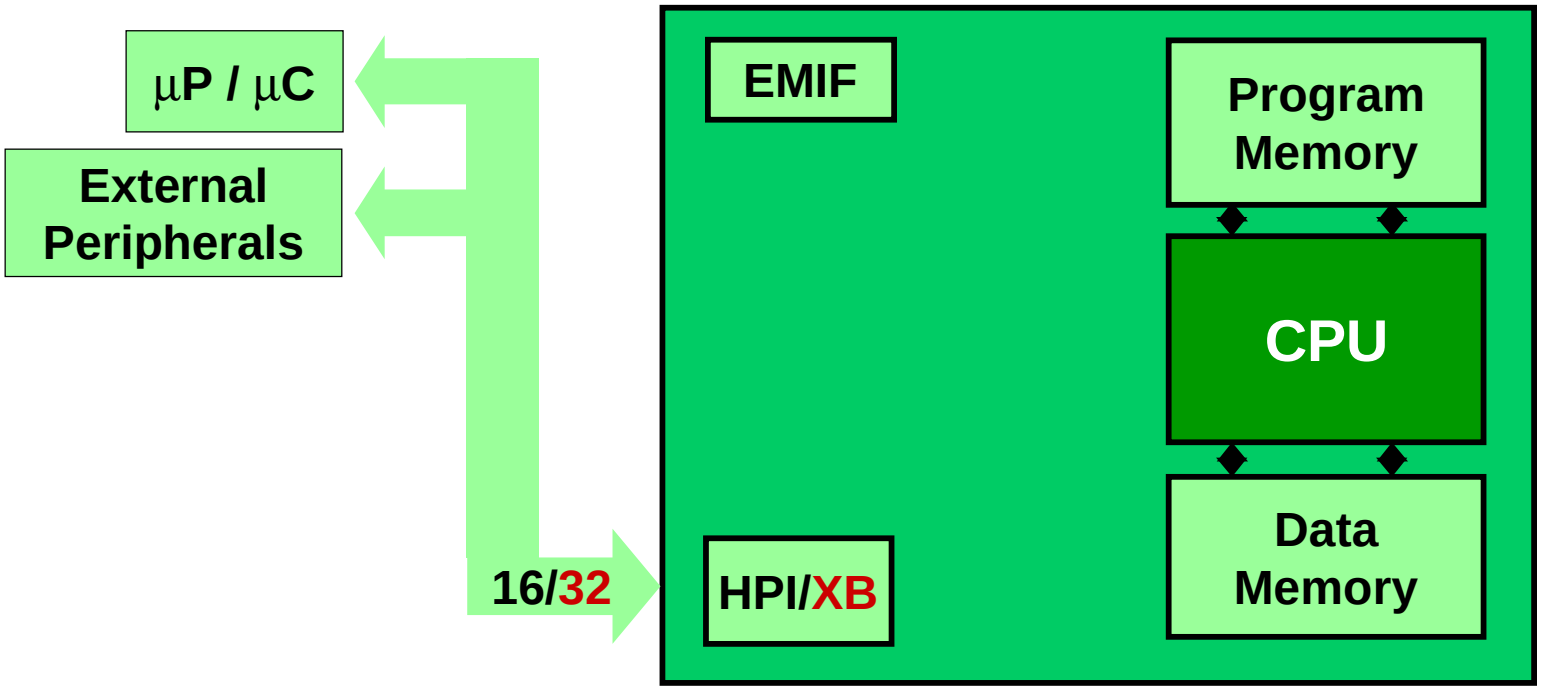
ASYNC

Provides I/F to ROMs, asynchronous RAMs and other parallel peripherals

EMIF: Glueless I/F

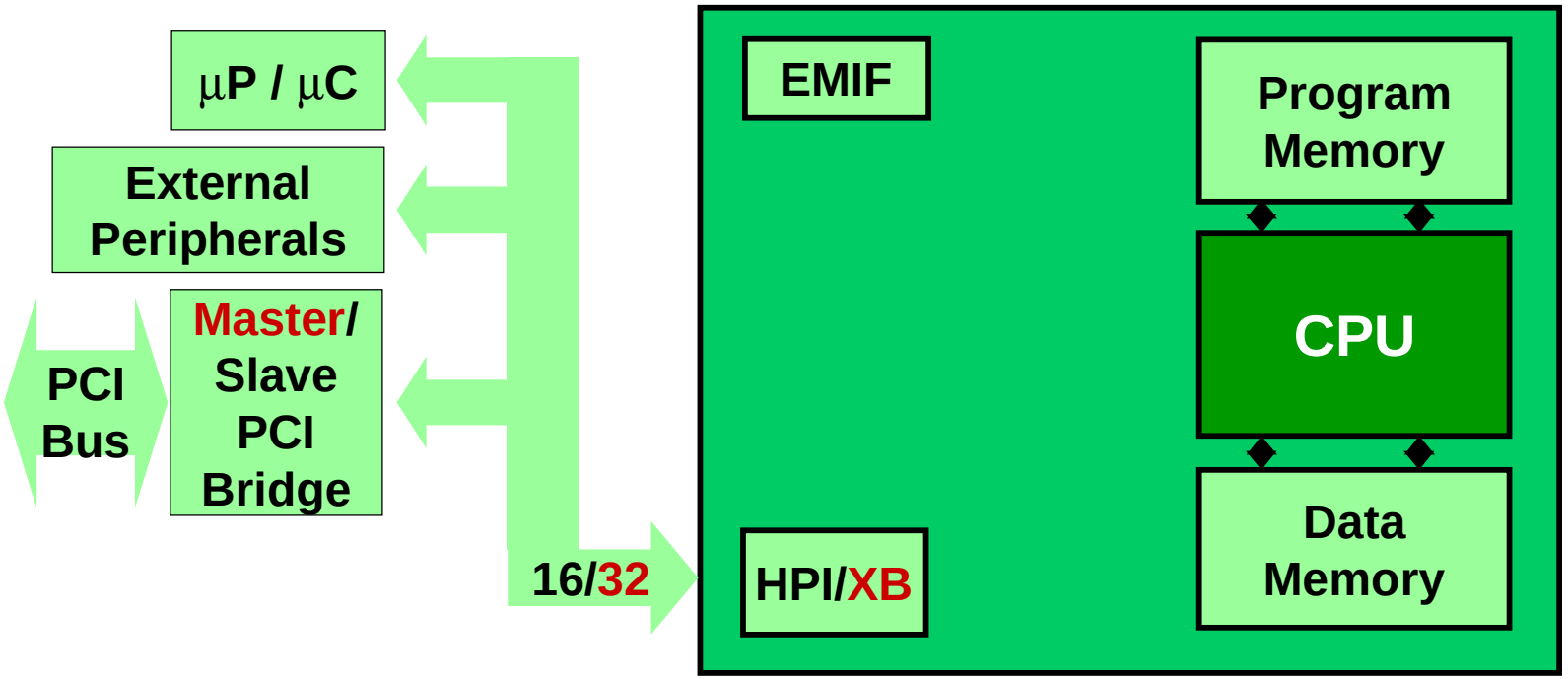
- **Glueless SDRAM I/F**
  - Decreasing costs because of popularity
  - Runs at 1/2 CPU clock rate
  - Can provide flexible refresh timing
- **Glueless SBSRAM I/F**
  - Supports: Pipelined, early write
- **Glueless Asynchronous I/F**
  - Software wait state timing
  - Hardware ready input
  - Non-pipelined access
  - Interface with Asynchronous RAMs, ROMs, FLASH, FIFOs

Interfacing: HPI and Expansion Bus



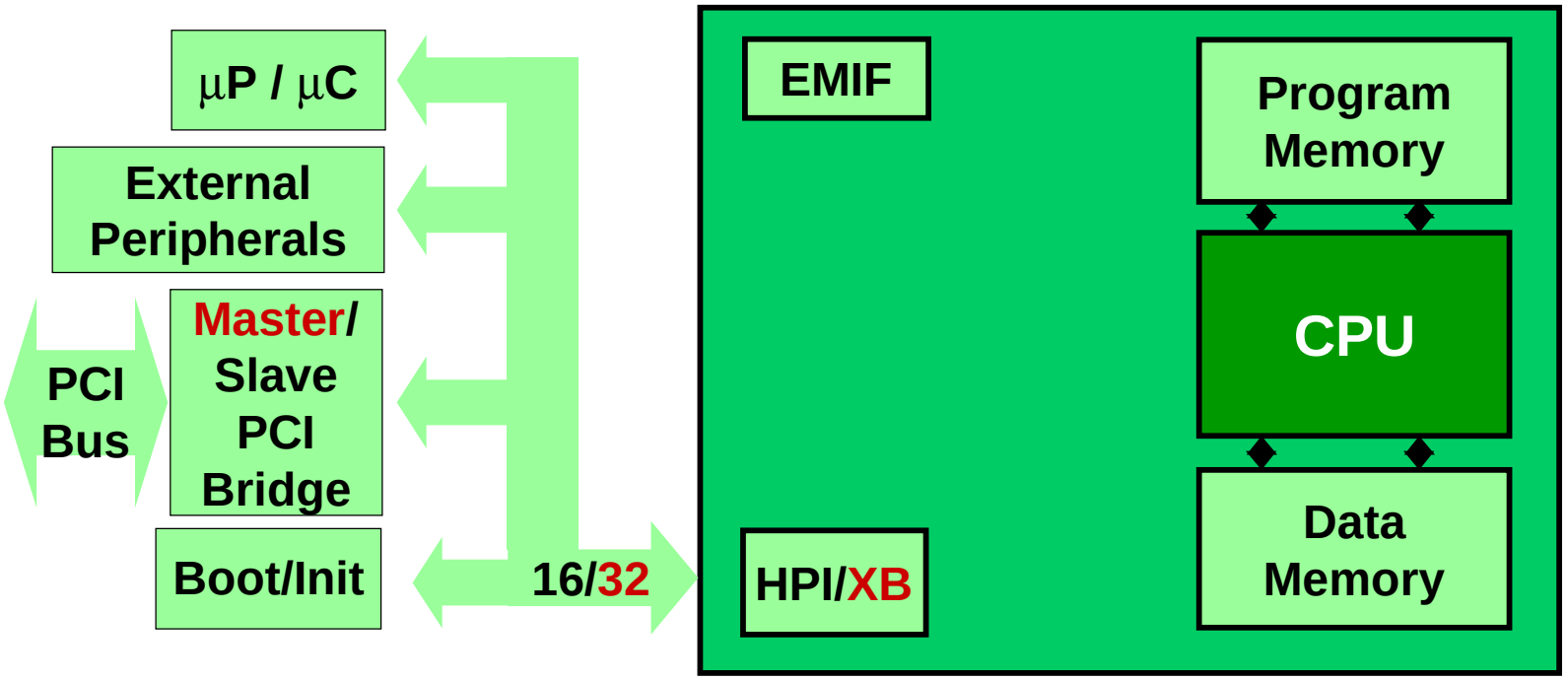
Connect to host micro or external peripherals through 16-bit HPI or 32-bit Expansion Bus (XB)

Interfacing: HPI and Expansion Bus



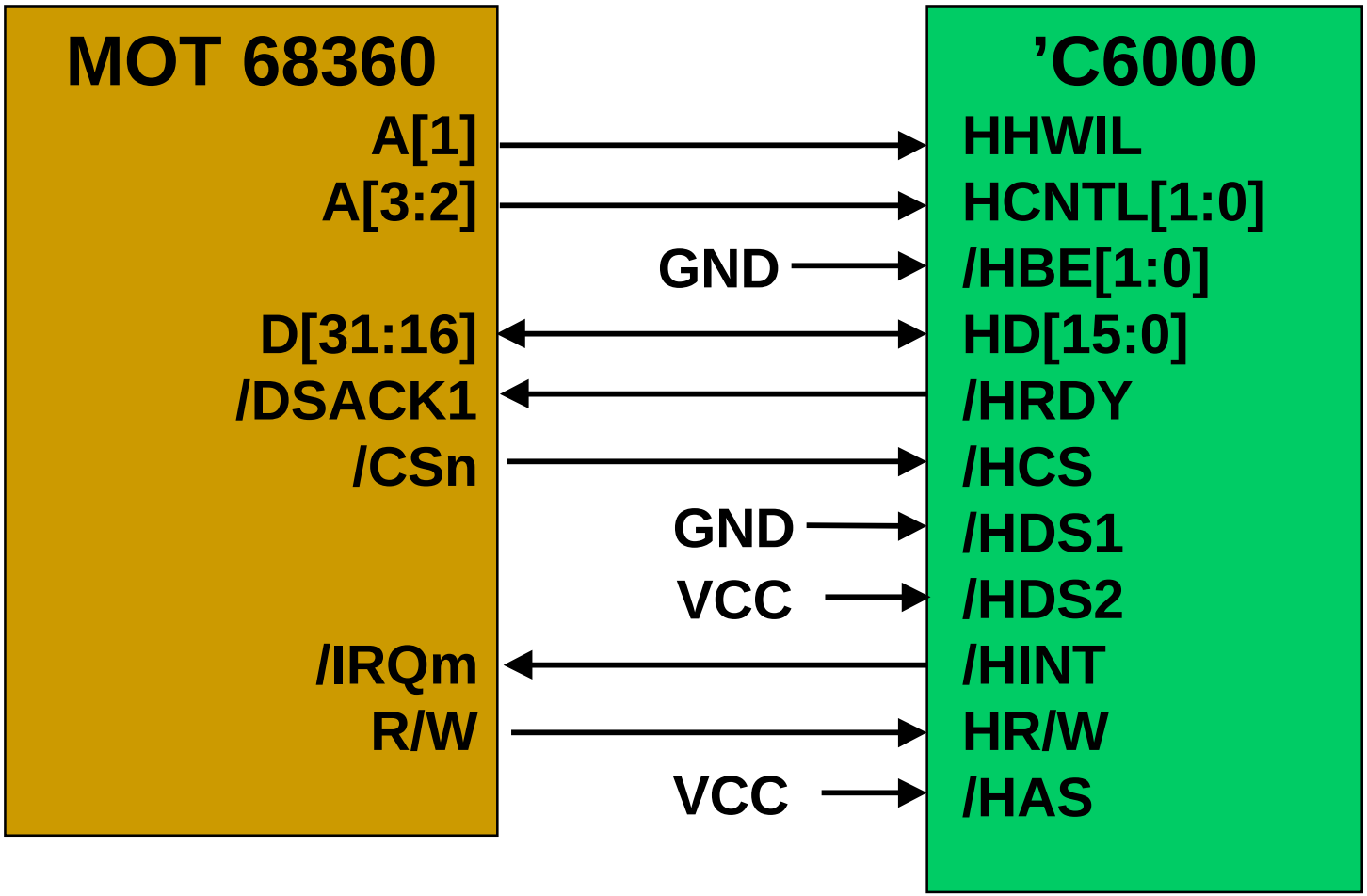
HPI provides convenient PCI slave connection while XB provides 32-bit, low-glue Master/Slave connection

Interfacing: HPI and Expansion Bus



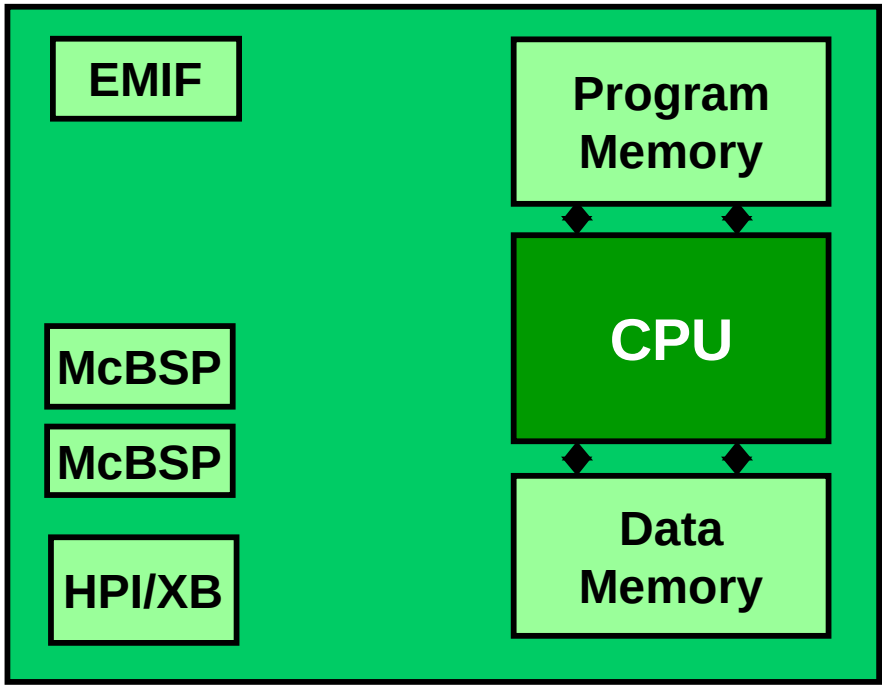
HPI/XB allows simple processor initialization (boot-up)

HPI: MOT 68360-to-'C6000



Interfacing: Multi-channel BSP

- Full duplex
- Runs at up to 1/2 CPU clock rate
- Double-buffered transmit, triple-buffered receive
- u-Law, A-Law companding
- Support for 128 time slots
- Compliant with variety of standards

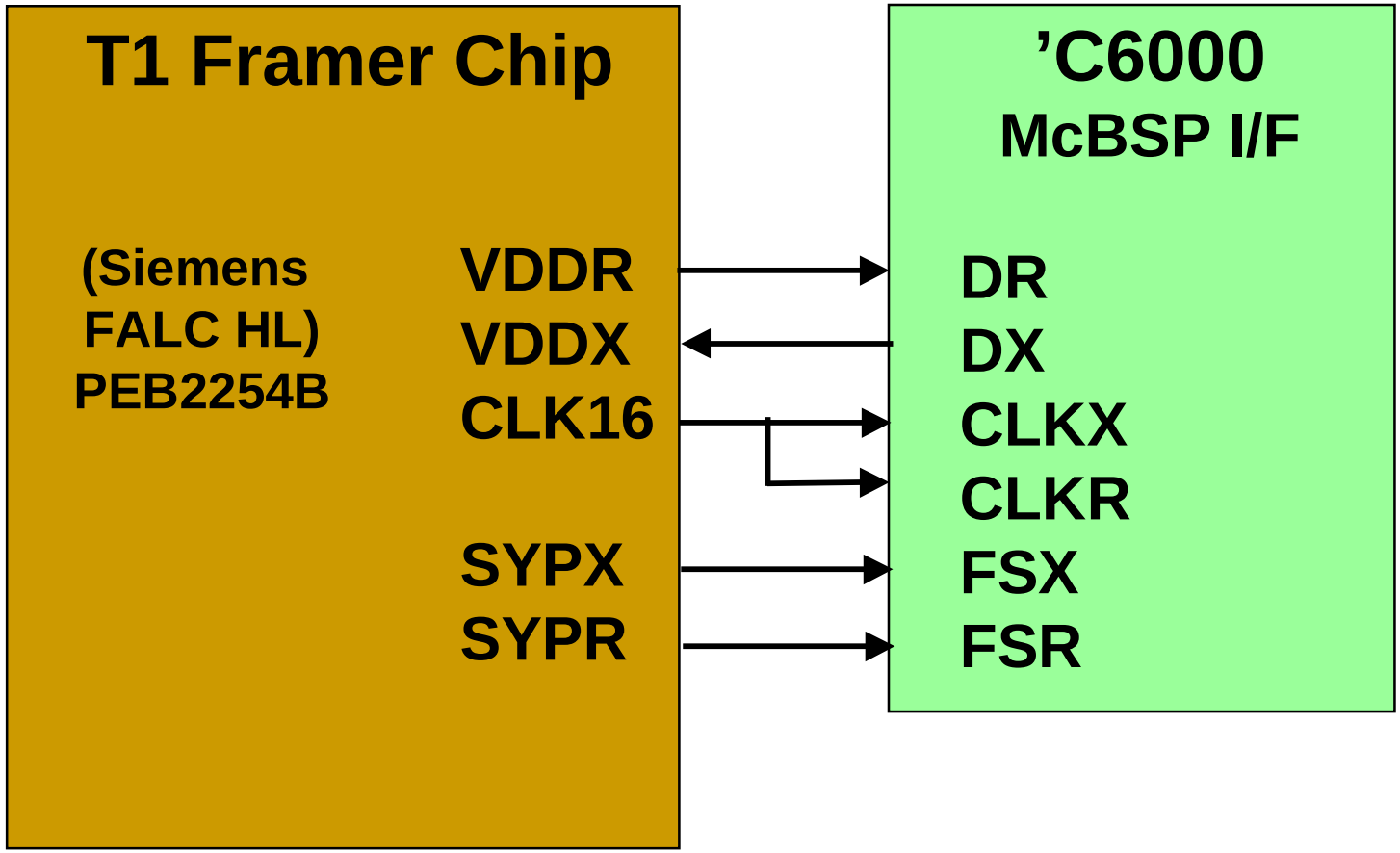


Interfacing: Multi-channel BSP

We support:

- ST-BUS
- T1/E1 Framing Chips
- IOM2
- SPI (4 different polarity bit-delay options)
- IIS
- AC97
- Industry-standard Codecs
- MVIP
- H.100
- Analog Interface Chips: TI TLC320

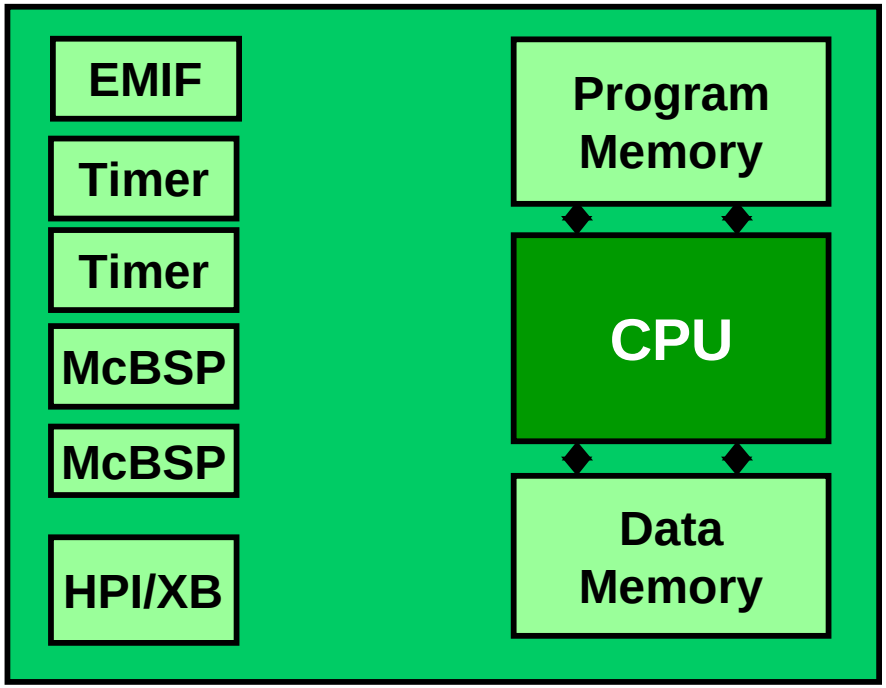
McBSP: Glueless I/F



Timers: General purpose

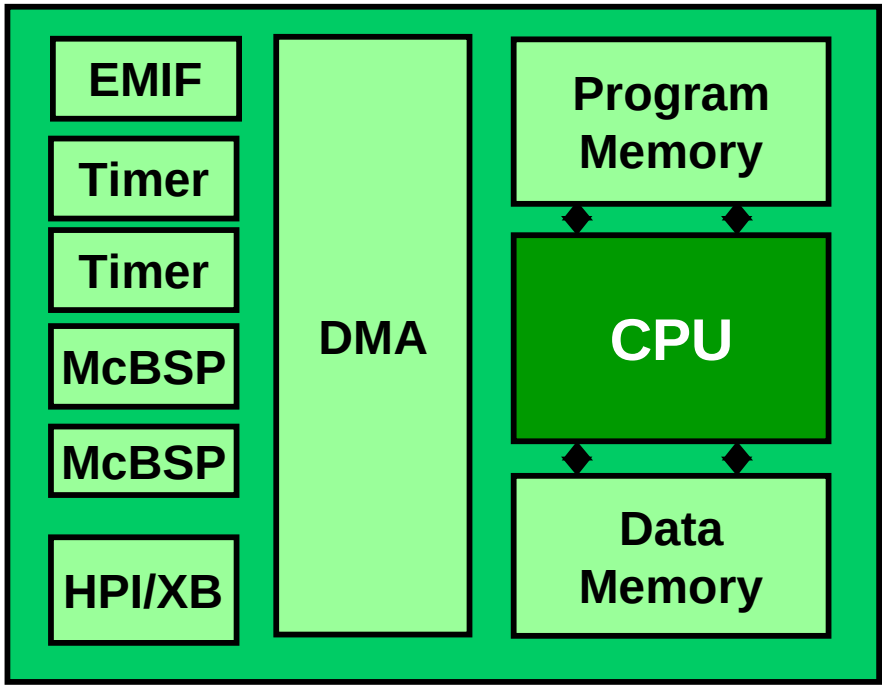
32-bit timers:

- Time events
- Count events
- Generate pulses
- Interrupt the CPU and send synchronization events to the DMA



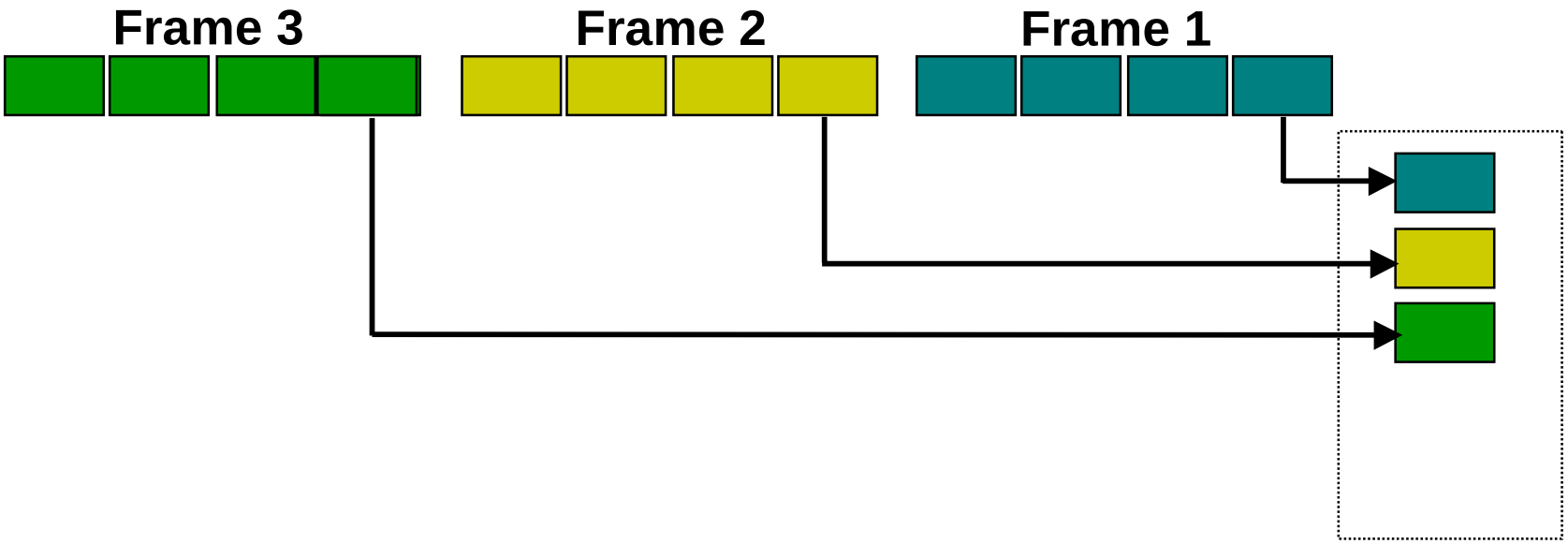
DMA: Transfers transparent to CPU

- DMA an access all components for transfers
- DMA transfers are transparent to the CPU.



DMA: Flexible addressing

Channel sorting handled by ’C6000 DMA



In Memory
(sorted by DMA)

EDMA: Enhanced DMA

	DMA	EDMA
Channels	4 channels + 1 dedicated to HPI	16 channels
Auto-Init	~2 tasks	69 tasks
Priority	4 fixed levels	2 variable levels

How do I work with TI's 'C6000?

What performance can I expect?

How do I get my performance?

How do I interface easily?

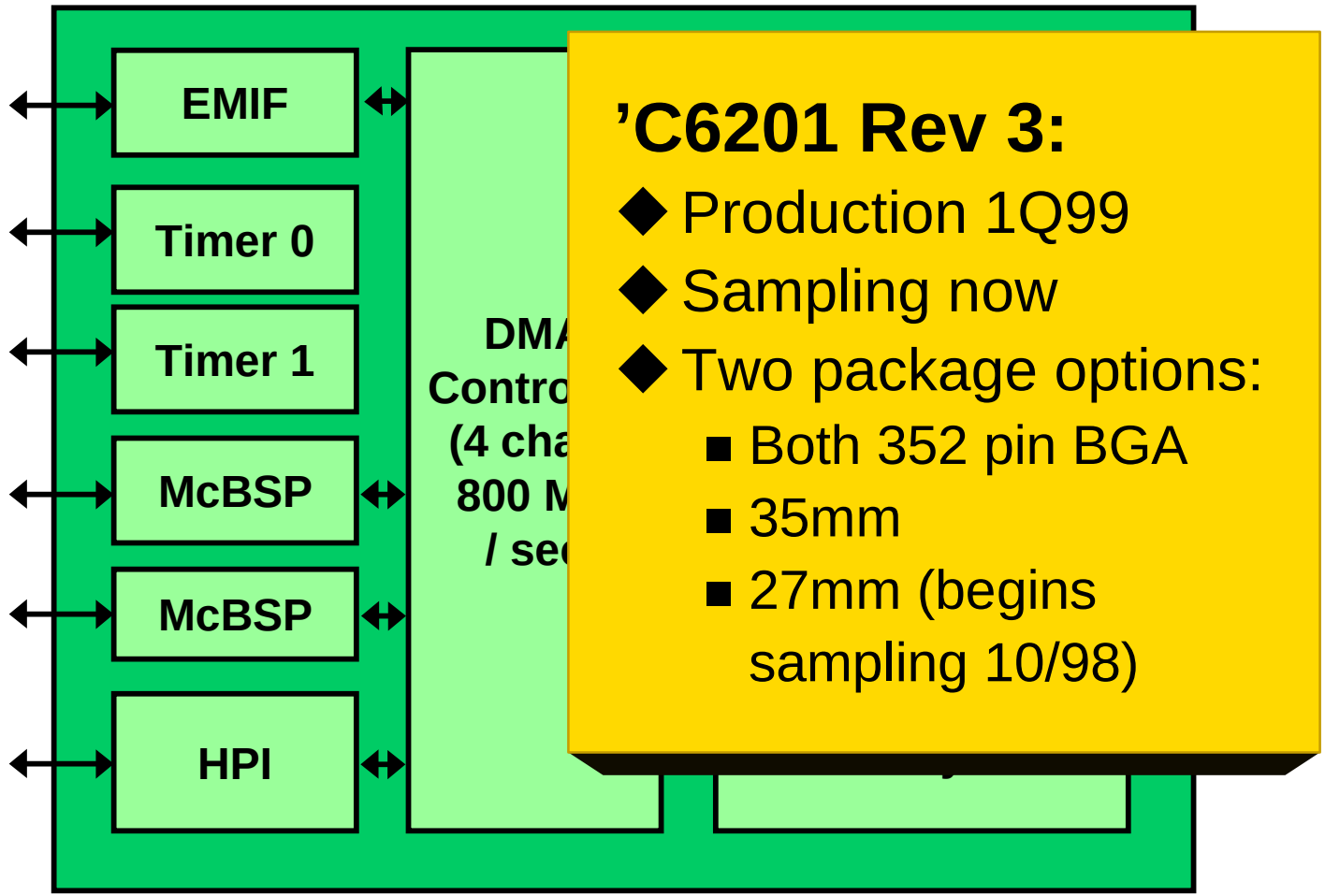
What are the new 'C6000 devices?

What is my power consumption?

How does TI enable maximum performance at lower cost?

'C6201: 1600 MIPS in production

TMS320C6201B Digital Signal Processor: 200 MHz at .18μ

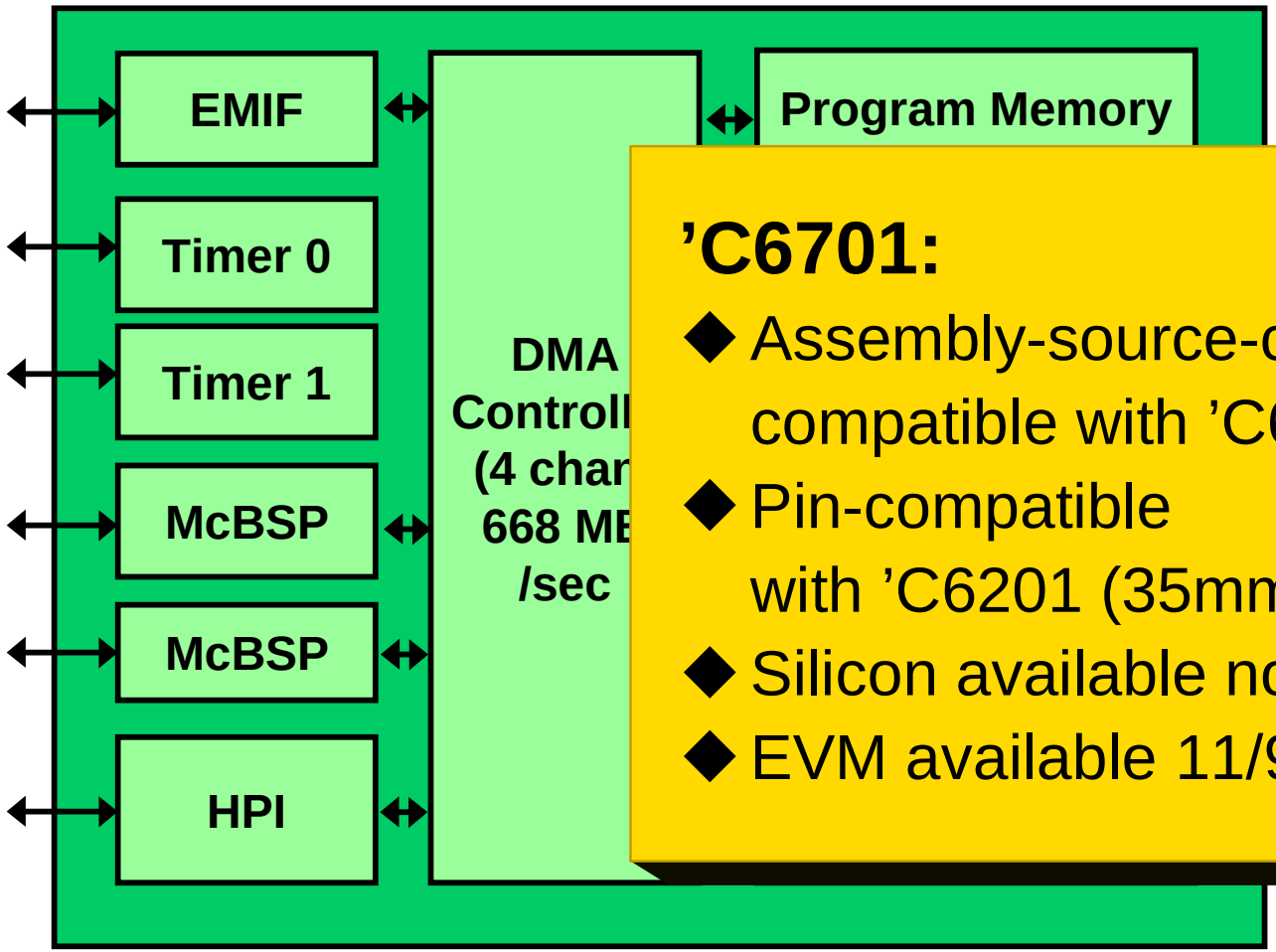


'C6201 Rev 3:

- Production 1Q99
- Sampling now
- Two package options:
  - Both 352 pin BGA
  - 35mm
  - 27mm (begins sampling 10/98)

'C6701: Floating point sampling now

TMS320C6701 Digital Signal Processor: 1 GFLOPS; 167 MHz at .18µ

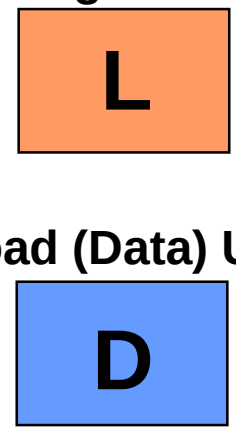


'C6701:

- Assembly-source-code compatible with 'C62x
- Pin-compatible with 'C6201 (35mm pkg.)
- Silicon available now
- EVM available 11/98

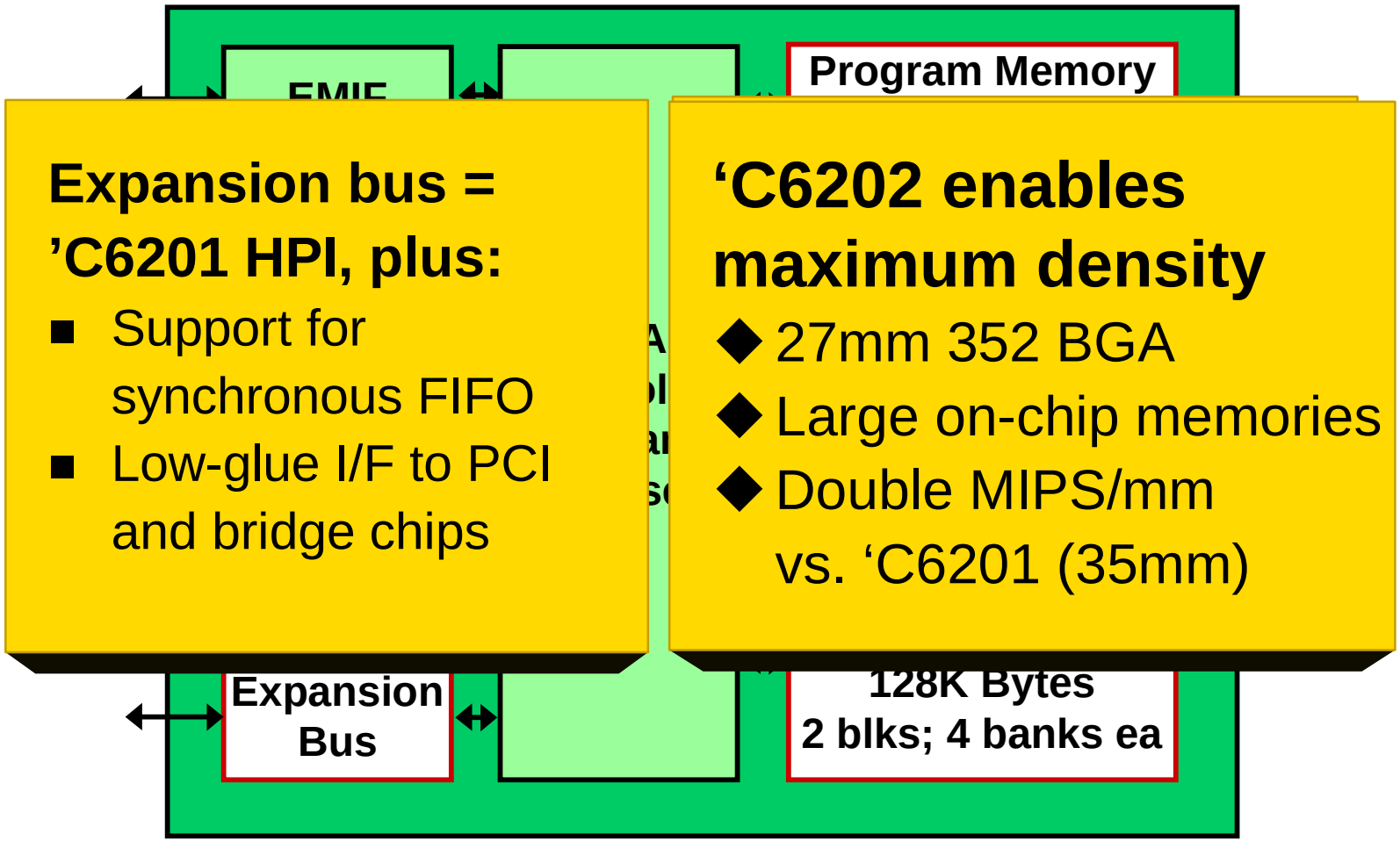
'C6000: Shared core

	'C62x Fixed Point	'C67x Floating Point
Shifter Unit	32-bit ALU shifts bit field operations branches	float comparison float ABS reciprocal sq. root reciprocal sngl/dbl prec. conv.
Multiply Unit	16x16 integer multiply	32x32 integer/floating mpy 64 x64 floating mpy
Logic Unit	40-bit ALU integer comparison normalization leftmost bit detect	fix/float conversion float arithmetic
Load (Data) Unit	loads and stores 32-bit add/sub address calculations	doubleword loads



'C6202: 2000 MIPS just announced

TMS320C6202 Digital Signal Processor: 250 MHz at .18µ



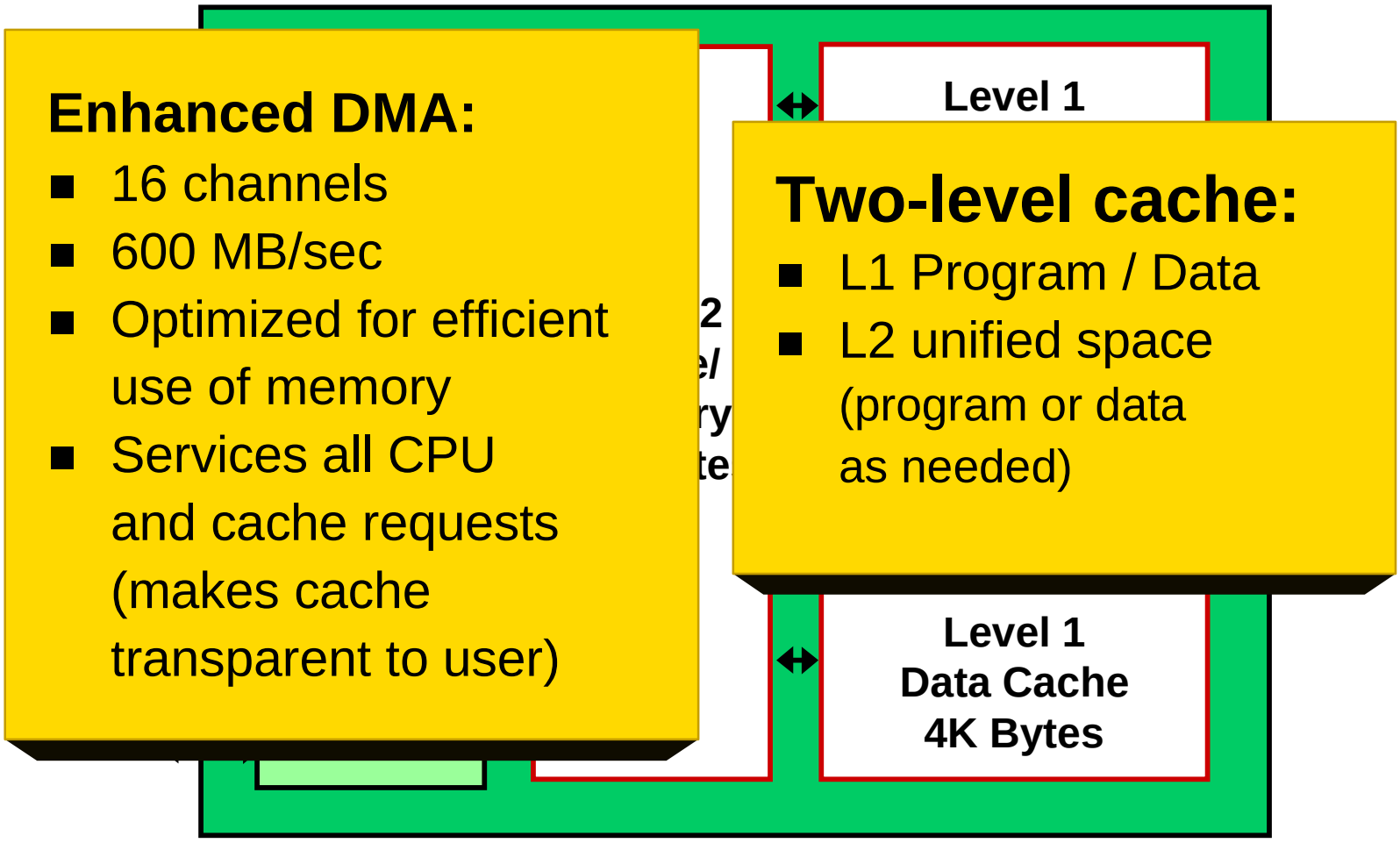
'C6202: Multi-channel = single-chip

Three vocoders + LEC + DTMF = One chip solution

Algorithm	Program Memory (Kbytes)	24-Channel Data Memory (Kbytes)
G.723	104	63.5
G.729A	90	69.5
G.726	4.8	5.6
32ms LEC	4.4	24.6
DTMF	5	10.8

‘C6211: $25-1200 MIPS just announced

TMS320C6211 Digital Signal Processor: 150 MHz at .18μ

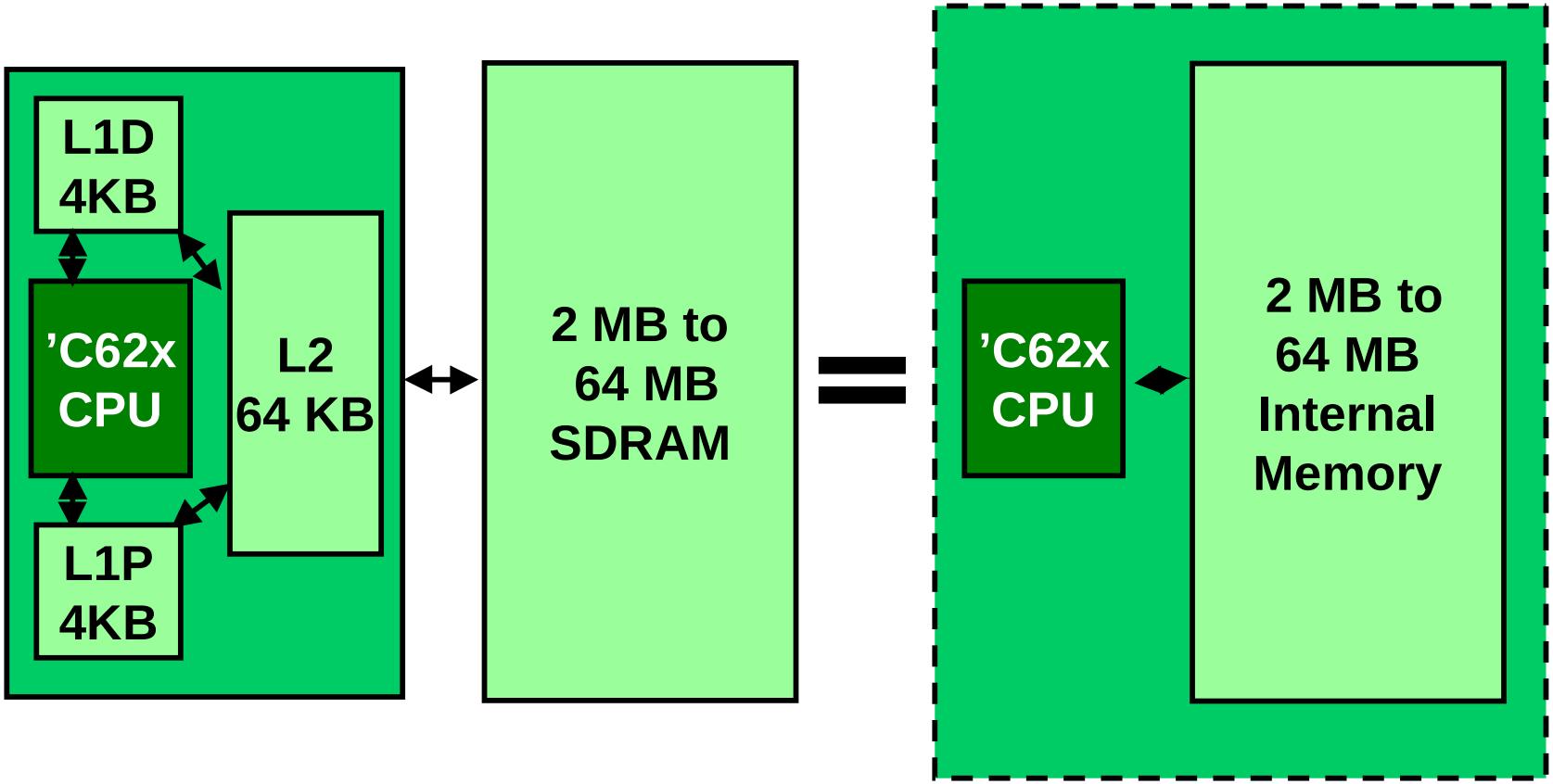


Internal Memory: Feeding the CPU

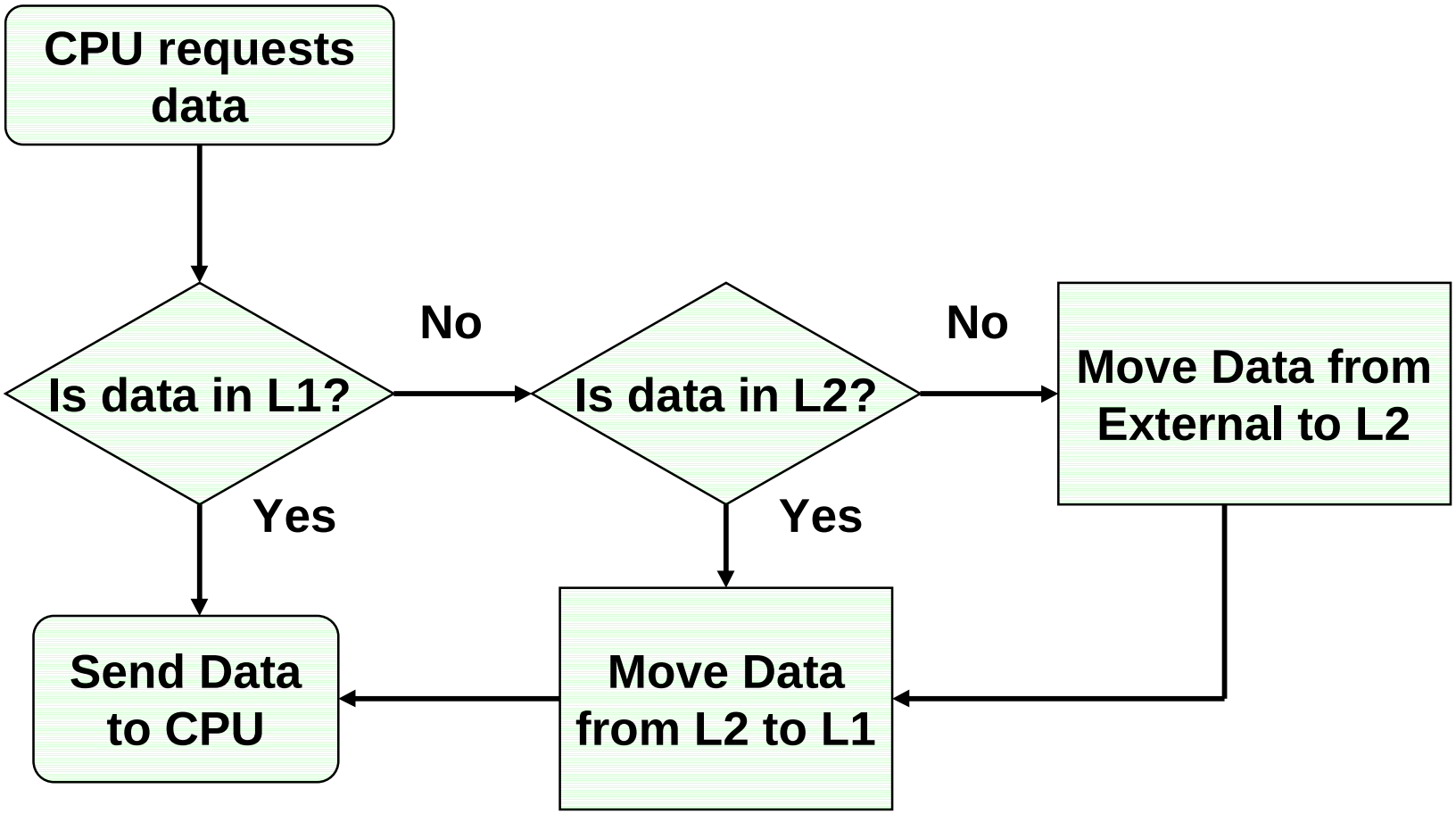
	L1 Memory		L2 Memory
	Program Memory	**Data Memory**	
'C6201	**64KB** 1 blk Pgm/Cache	**64KB** 2 blks 4 banks ea	**External**
'C6701	**64KB** 1 blk Pgm/Cache	**64KB** 2 blks 8 banks ea	**External**
'C6202	**256KB** 1 blk Pgm/Cache 1 blk Mapped Pgm	**128 KB** 2 blks 4 banks ea	**External**
'C6211	**4 KB** 1 blk Cache	**4 KB** 1 blk Cache	**64 KB** 4 blk Mapped Cache

'C6211: Turns KB into MB

'C6211 Two-Level Cache turns
72 KB cache into 2 MB - 64 MB on-chip memory

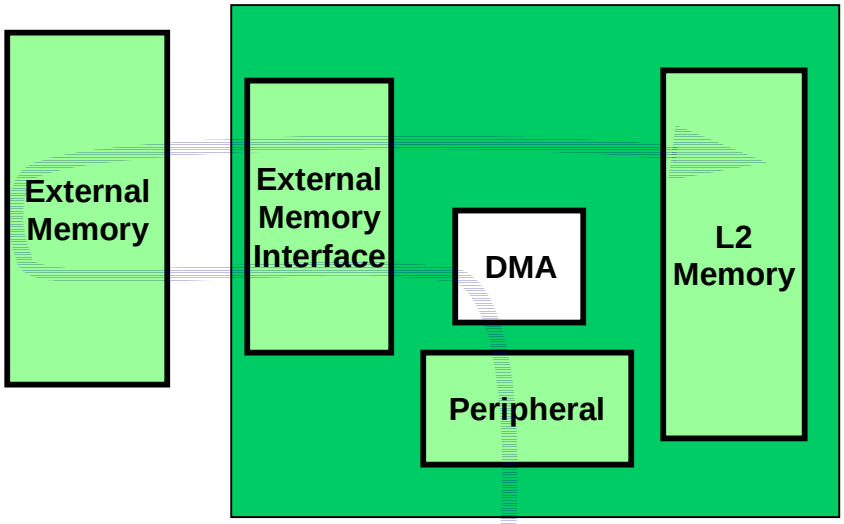


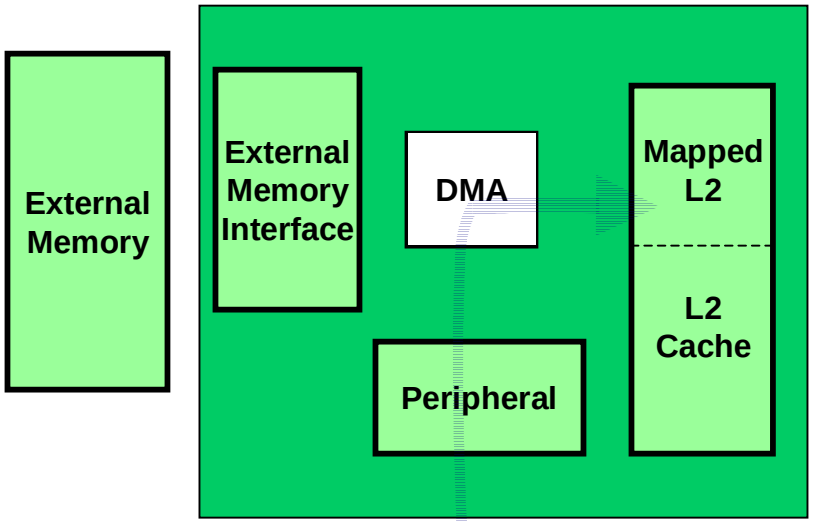
'C6211: L1/L2 cache flow



'C6211: L2 flexibility

- **Typical cache architecture**
  - Lacks non-cacheable regions
  - Requires external storage of peripheral data
- **'C6211**
  - Configurable as cache and direct mapped
  - Allows peripheral data storage on-chip





'C6211: L1/L2 cache benchmarks

Performance relative to 'C62x infinite on-chip memory

Cycle count performance

How do I work with TI's 'C6000?

What performance can I expect?

How do I get my performance?

How do I interface easily?

What are the new 'C6000 devices?

What is my power consumption?

How does TI enable maximum performance at lower cost?

'C6000: Power consumption

Power consumption is a product of:

- ✔ Internal memory accesses
- ✔ Functional units running per cycle
- ✔ External memory accesses
- ✔ Efficient process technology

… 'C6000 produces up to 10x the industry's highest raw performance with industry-standard or better consumption levels.

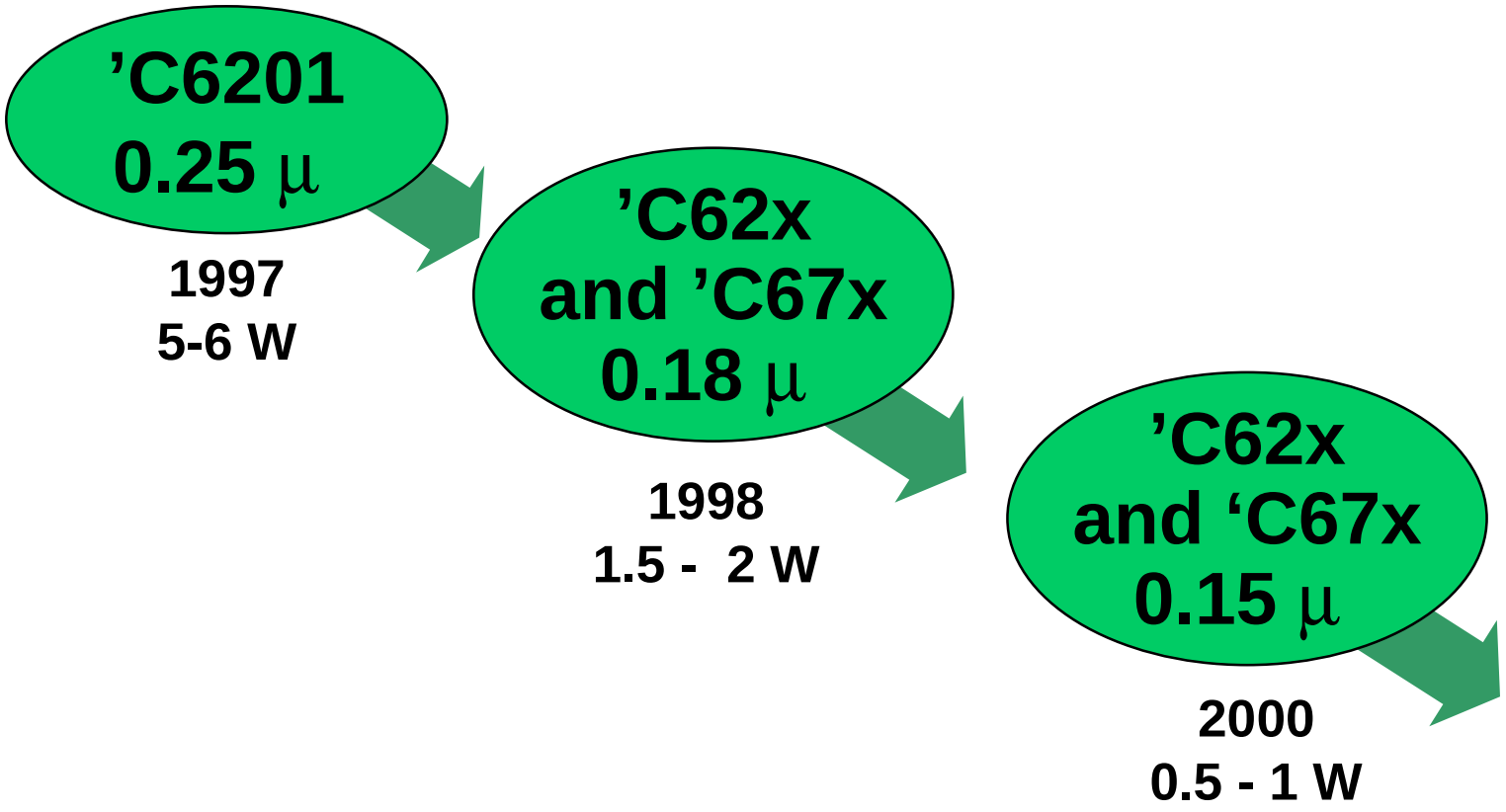
<1.9W at industry-standard activity

Industry-standard power measurements

- 50/50 activity
- CPU & memory only

		50% High 50% Low 'C6201B		75% High 25% Low 'C6201B	
		% of Total	W @ 200 MHz	% of Total	W @ 200 MHz
CPU and Memory	CPU	17%	0.42	18%	0.49
	Internal memory	8%	0.18	8%	0.22
	Clocking	44%	1.07	39%	1.07
	Total	69%	**1.7**	66%	**1.8**
Peripherals	DMA/EMIF	3%	0.07	3%	0.09
	DMA memory access	9%	0.21	12%	0.31
	Other	0%	0.01	0%	0.01
	Total	12%	0.3	15%	0.4
Core VDD (2.5 or 1.8V total)		81%	**2.0**	81%	**2.2**
I/O (DVDD) Total		19%	0.4	19%	0.5
TOTAL		100%	**2.4**	100%	**2.7**

'C6000: Lower power consumption



Typical 200-MHz CPU and memory power consumption

'C6000: Power down modes

- **Mode 1**
  Shut off CPU clocks
  - Can be revived through peripheral or external interrupts
- **Mode 2**
  Shut off all CPU clocks / Leave PLL on
  - Can be revived through reset
- **Mode 3**
  Shut off all CPU clocks and PLL
  - Can be revived through reset

How do I work with TI's 'C6000?

What performance can I expect?

How do I get my performance?

How do I interface easily?

What are the new 'C6000 devices?

What is my power consumption?

How does TI enable maximum performance at lower cost?

'C6202: Lowest $ per channel

	'C6202 Channels @ 250 MHz	'C6202 @ $130 ea $/Chn	Typical Channels @ 100 MHz	Typical* @ $30 ea $/Chn
G.729A	36+	<$ 3.61	6	$ 5.00
G.723.1	28+	<$ 4.64	5	$ 6.00
G.726	60-100	$ 2.16 - 1.30	10+	<$ 3.00
Echo Cancl. (32-msec tail)	60+	<$ 2.17	10	$ 3.00
GSM EFR	20+	<$ 6.50	4-6	$ 7.50
V.34/V.90	15-18	$ 8.66	3	$10.00

* Large on-chip RAM device

'C6701: Lowest $ per MFLOPS

200 MMACS for $300*

ADI-21160
667 MFLOPS

OR

334 MMACS for $165

'C6701
1 GFLOPS

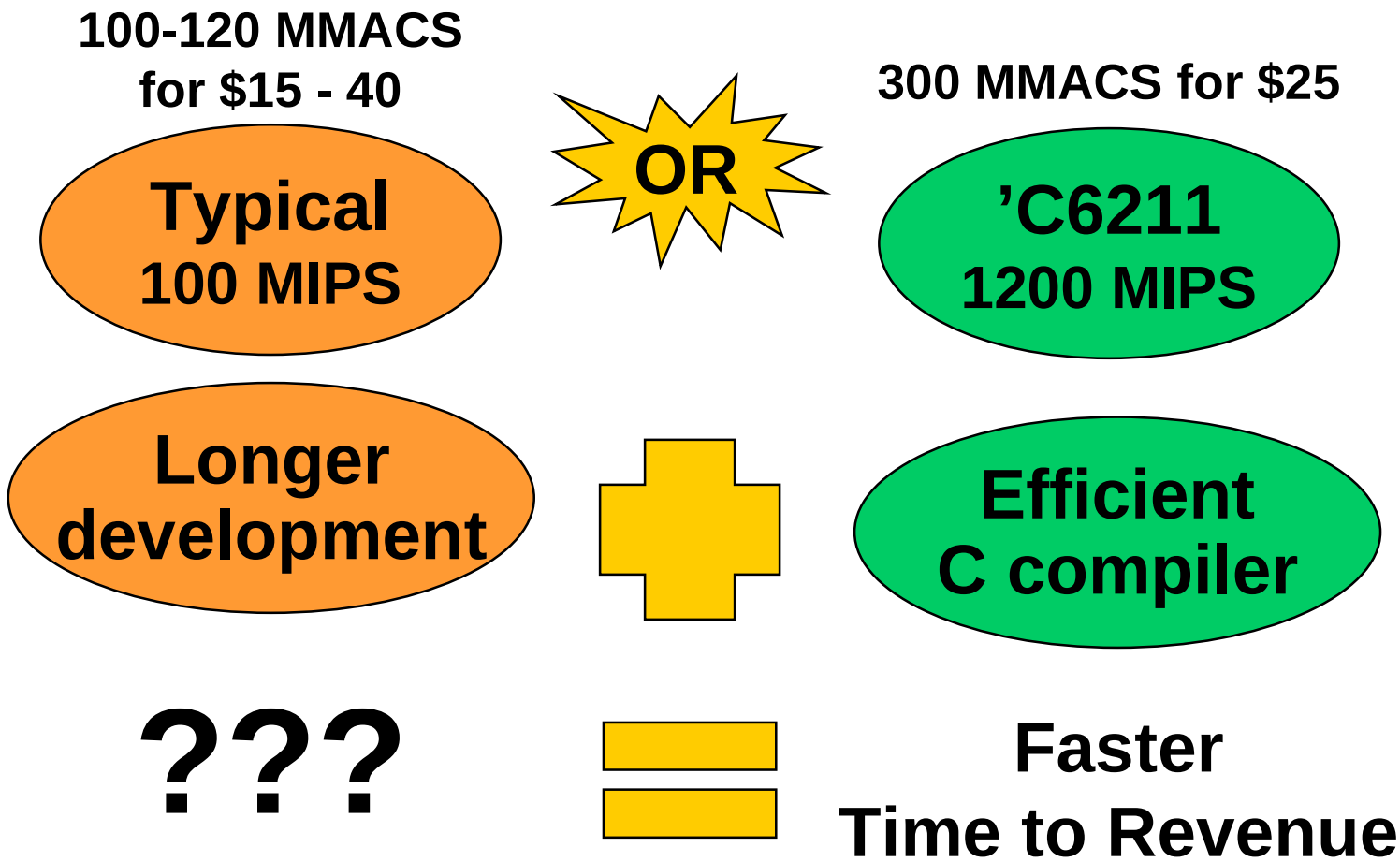
'C6701 floating-point silicon <u>available today</u>:

- Ultra-high precision
- Development in C code, with highly efficient tools available today
- Assembly-source-code compatible with 'C62x fixed-point devices

TI and ADI sample pricing

*Source: ADI News Release 6/22/98.

'C6211: Greatest value per $25



'C6000: Universal platform