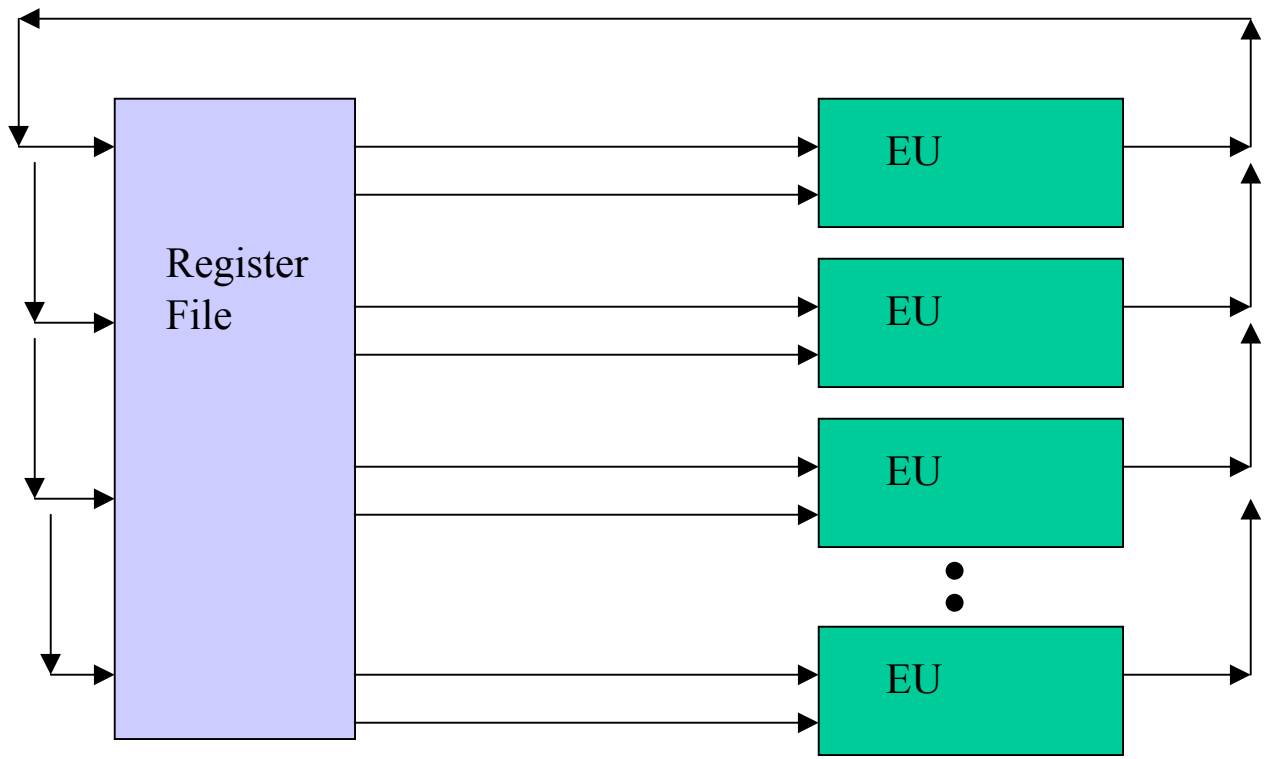
VLIW Architectures

- Very Long Instruction Word Architecture
  - ⇒One instruction specifies multiple operations
  - ⇒All scheduling of execution units is static
    - →Done by compiler
  - ⇒Static scheduling should mean less control, higher clock speed. Less control means more room for execution units.
- Currently very popular architecture in embedded applications
  - ⇒DSP, Multimedia applications
  - ⇒No compiled legacy code to support, all code libraries in some form of high level language

Keeping Execution Units Busy

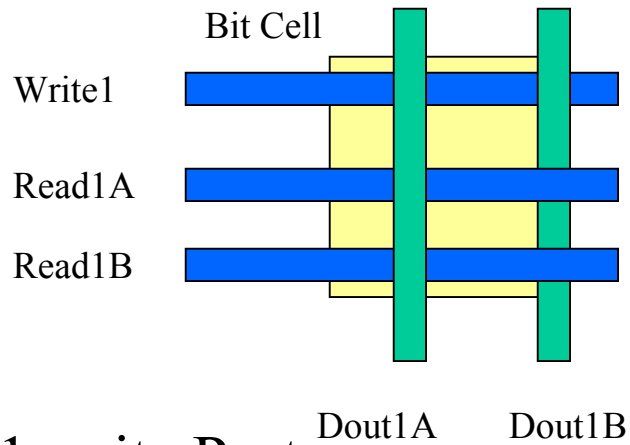
- Execution units are 2-input, 1-output blocks (typically)
- Each clock cycle, need to read 2N operands, write N results for N Execution Units



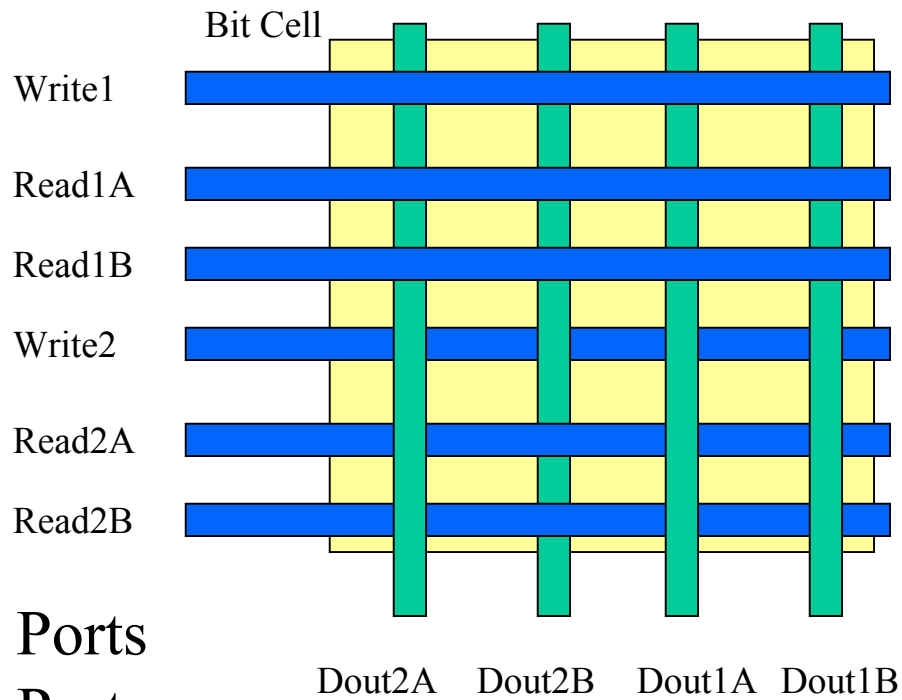
Multi-Ported Register File Design has Limits

- Area of the register file grows approximately with the square of the number of ports

  ⇒Typically routing limited, each new port requires adding new routing in both X and Y direction



1 write Port
2 Read Ports



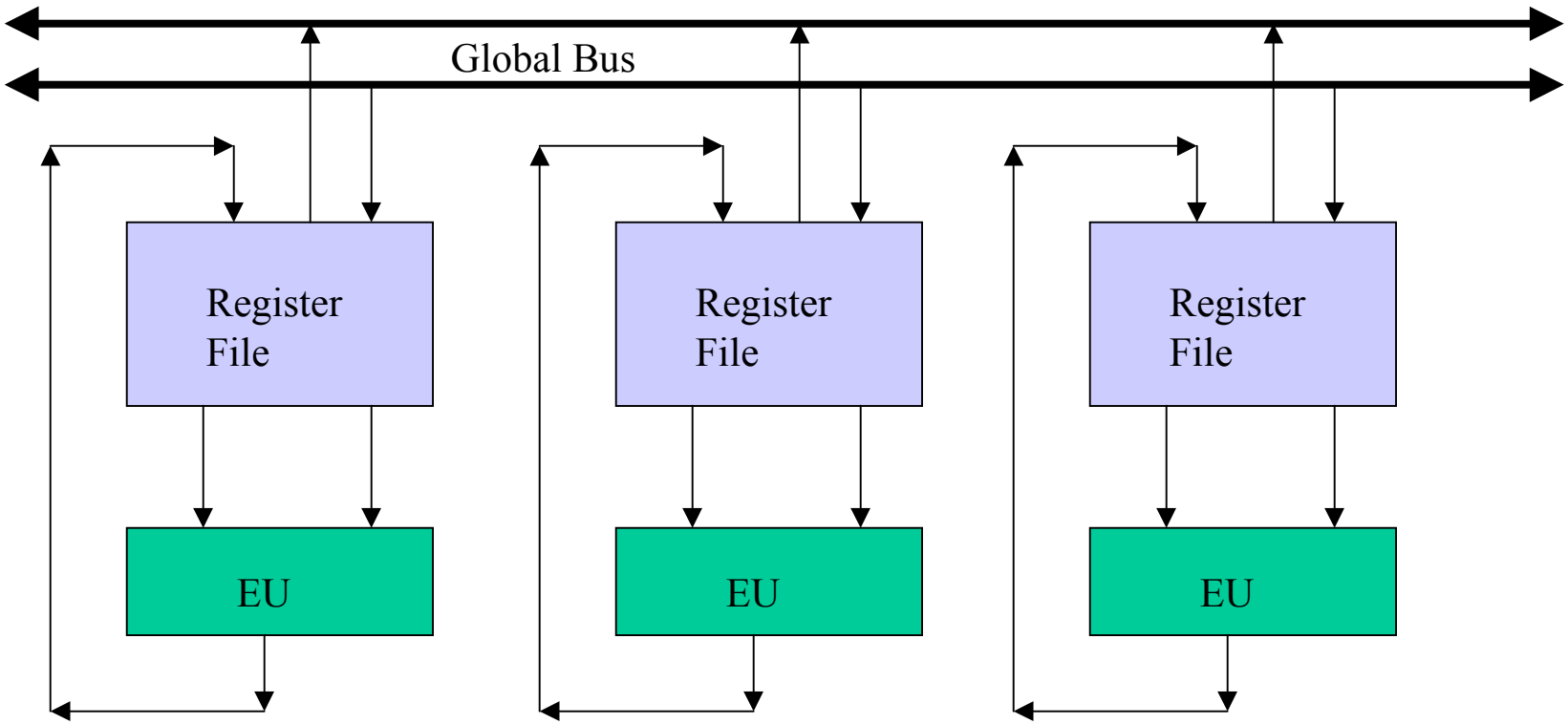
2 write Ports
4 Read Ports

Multiported Register Files (cont)

- Read Access time of a register file grows approximately linearly with the number of ports
  - ⇒Internal Bit Cell loading becomes larger
  - ⇒Larger area of register file causes longer wire delays
- What is reasonable today in terms of number of ports?
  - ⇒Changes with technology, 15-20 ports is currently about the maximum (read ports + write ports)
  - ⇒Will support 5-7 execution units simultaneous operand accesses from register file

Solving the Register File Bottleneck

- Create partitioned register files connected to small numbers of Executions units (perhaps as many as one register file per EU)



Register File Communication

- Architecturally Invisible
  - ⇒Partitioned RFs appear as one large register file to the compiler
  - ⇒Copying between RFs is done by control
  - ⇒Detection of when copying is needed can be complicated; goes against VLIW philosophy of minimal control overhead
- Architecturally Visible, have Remote and Local versions of instructions
  - ⇒Remote instructions have one or operands in non-local RF
  - ⇒Copying of remote operands to local RFs takes clock cycles
  - ⇒Because copying is ‘atomic’ part of remote instruction, execution unit is idle while copying is done => performance loss.

Register File Communication (cont).

- Architecturally Visible, have explicit copy operations

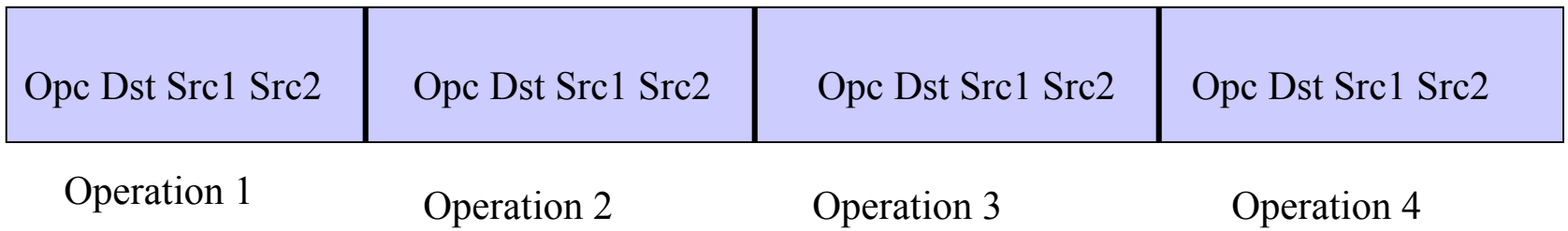
  ⇒Separation of copy and execution allows more flexible scheduling by compiler

```
move   r1, r60            (r60 in another RF)
independent instr a   (cycles for copy to complete)
independent instr b  (cycles for copy to complete)
add   r2, r1, r3
```

Instruction Compression

- Embedded Processors often put a premium on code size

  ⇒Uncompressed VLIW instructions are wide (of course!)



- How we reduce word length?

  ⇒NOPs are common, use only a few bits (2-3) to represent a NOP

When are instructions decompressed?

- On Instruction Cache (ICache) fill
  - ⇒Cache fill is a slow operation to begin with; limited by speed of external memory bus
  - ⇒Compression algorithm can be more complicated because have more time to perform the operation
  - ⇒ICache has to hold uncompressed instructions - limits cache size
- On instruction fetch
  - ⇒ICache holds compressed instructions
  - ⇒Decompression in critical path of fetch stage, may have to add one or more pipeline stages just for decompression

Importance of the Compiler

- The quality of the compiler will determine how much of the potential performance of a VLIW architecture is actually realized.

  $\Rightarrow$When MFLOP figures are specified for VLIW architectures, these are usually for the theoretical performance of the architecture. Actual performance can be quite lower.

- Because of the dependence of the compiler on the hardware, new versions of the architectures can force major rewrites of the compiler - very costly
- Often the user has to place hints in the high-level code (‘pragmas’) that help the compiler produce more optimal code.

TMS320C6X CPU

- 8 Independent Execution units
  - ⇒Split into two identical datapaths, each contains the same four units (L, S, D, M)
- Execution unit types:
  - ⇒L : Integer adder, Logical, Bit Counting, FP adder, FP conversion
  - ⇒S : Integer adder, Logical, Bit Manipulation, Shifting, Constant, Branch/Control, FP compare, FP conversion, FP seed generation (for software division algorithm)
  - ⇒D : Integer adder, Load-Store
  - ⇒M : Integer Multiplier, FP multiplier
- Note that Integer additions can be done on 6 of 8 units!
  - ⇒Integer addition/subtraction is a very common operation!

TMS320C6X CPU (cont).

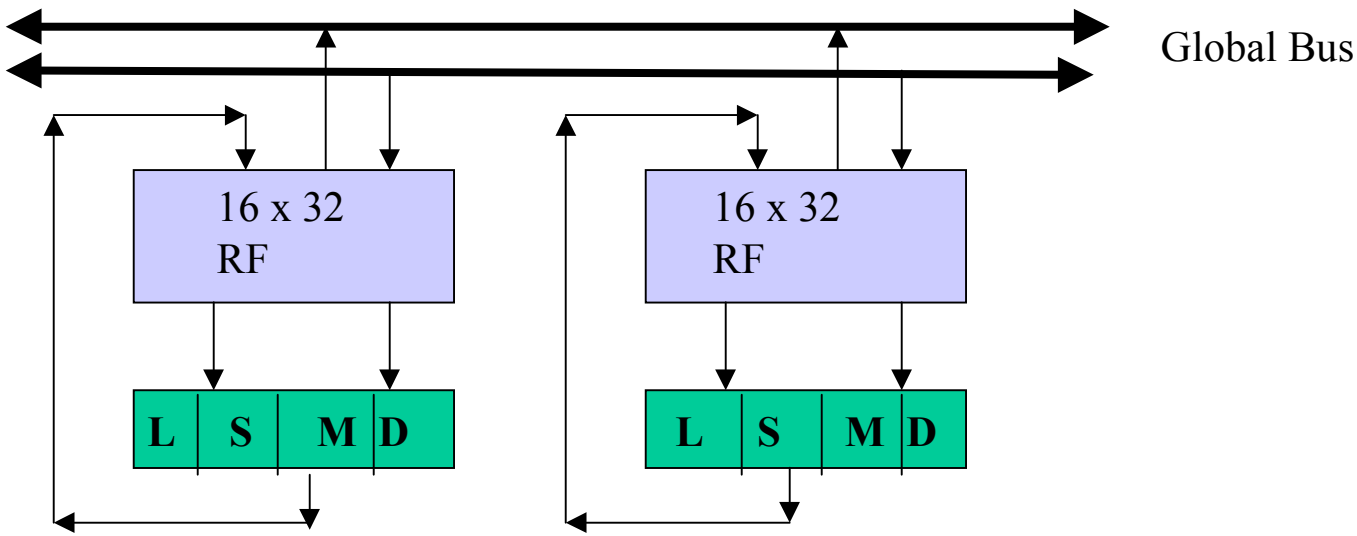
- Max clock speed of 200 Mhz

  ⇒Not too impressive compared to Intel - my guess is process limitations rather than design limitations

- Each datapath has a 16 x 32 Register file

  ⇒10 Read ports, six Write ports

  ⇒Can transfer values between the two register files



Instruction Encoding

- Internal Execution path is 256 bits
  - ⇒Each operation is 32 bits wide => 8 operations per clock
  - ⇒A fetch packet is a group of instructions fetched simultaneously. Fetch packet has 8 instructions.
  - ⇒A execute packet is a group of instructions beginning execution in parallel. Execute packet has 8 instructions.
- Instructions in ICache have an associated P-bit (Parallel-bit).
  - ⇒Fetch packet expanded to 1 to 8 Execute packets depending on P-bits

Fetch Packet to Execute Packet Expansion

Fetch Packet

A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0

P-bits, A-H executed serially

8 instructions

Execute Packet

n	n	A	n	n	n	n	n
n	B	n	n	n	n	n	n
n	n	n	n	n	C	n	n
n	n	n	n	n	D	n	n
n	n	n	E	n	n	n	n
F	n	n	n	n	n	n	n
n	n	n	n	n	n	G	n
n	n	n	n	n	n	n	H

64 instructions

Fetch Packet to Execute Packet Expansion (cont.)

Fetch Packet

A	B	C	D	E	F	G	H
1	1	0	1	0	0	1	0

P-bits

A||B||C, D||E, F, G||H

Execute Packet

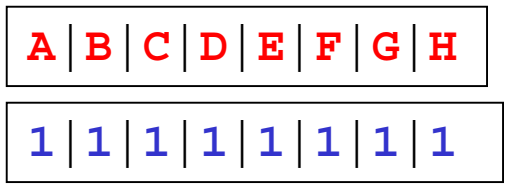
n	B	A	n	n	C	n	n
n	n	n	E	n	D	n	n
F	n	n	n	n	n	n	n
n	n	n	n	n	n	G	H

40 instructions

P-bit String of ‘1’s followed by ‘0’ means those execute in parallel. String starting with ‘0’ indicates sequential execution.

Fetch Packet to Execute Packet Expansion (cont.)

Fetch Packet



Execute Packet

8 instructions

P-bits

A||B||C||D||E||F||G||H

P-bit String of ‘1’s followed by ‘0’ means those execute in parallel. String starting with ‘0’ indicates sequential execution.

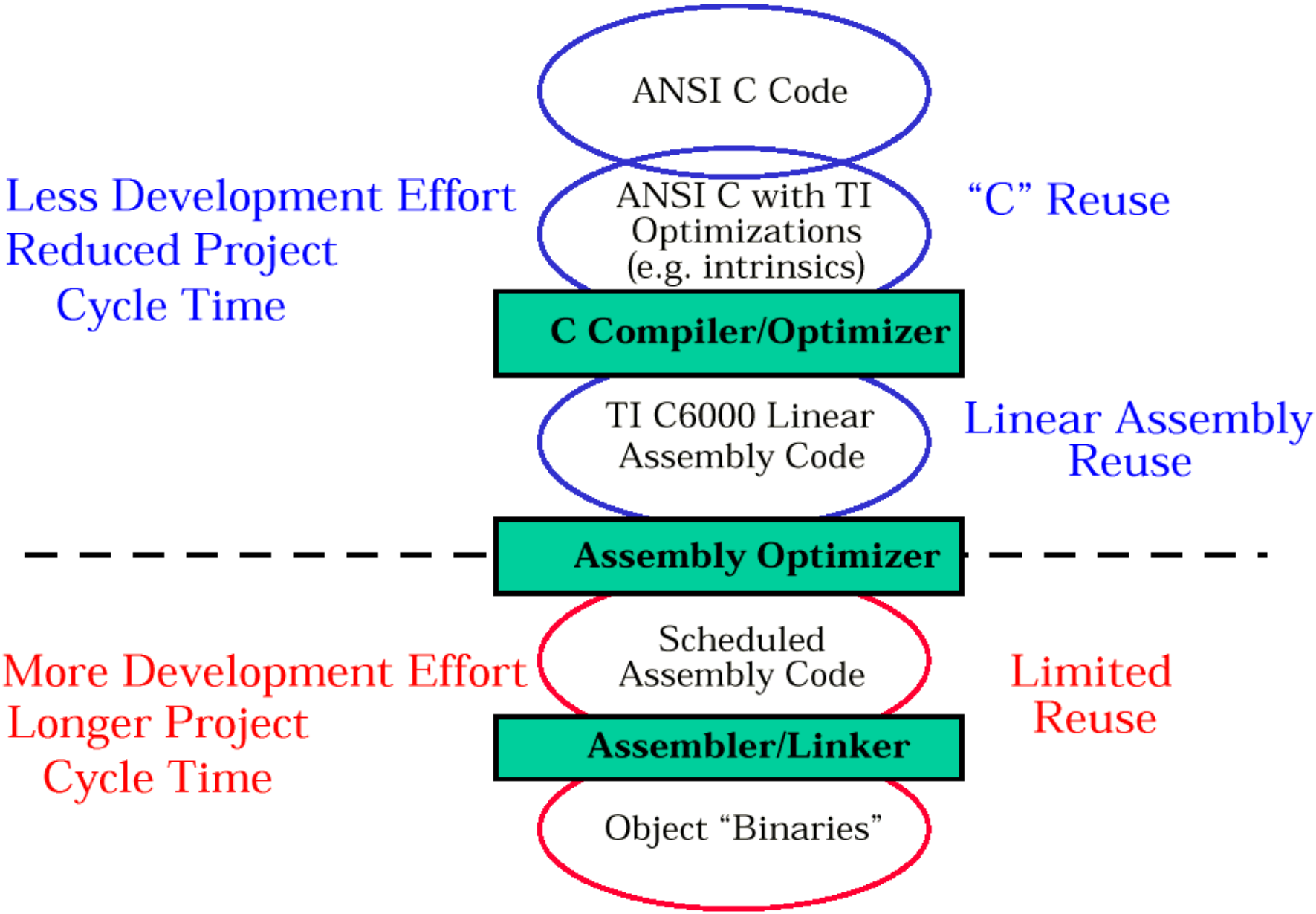
Pipeline

- Fetch - four phases
  - ⇒PG - program address generate
  - ⇒PS - program address send
  - ⇒PW - program access ready (cache access) - Memory stall is the only stall case in the pipeline
  - ⇒PR - program fetch packet receive
- Decode - two phases
  - ⇒ DP - instruction dispatch, convert fetch packet to execute packet expansion, routed to decode of functional units (functional units do multiple operations)
  - ⇒DC - instruction decode

Pipeline (cont)

- Execute - maximum of 10 phases
  - ⇒ 90% of the instructions only use first 5
  - ⇒ double precision FP add/mults use last 5
- Result Latency: number of execute phases used by an operation (most only use 1)
- Delay slots: Result Latency minus 1. If zero, then result is available for next execute packet.
  - ⇒If non-zero, then independent operations must scheduled in the delay slots
- Functional Unit Latency (repetition rate of Functional Unit)
  - ⇒Either 1, 2 or 4. (1 for common operations and LD/Stores)

Figure 1. TMS320C6000 Code Reuse Efficiency Diagram



Guidelines for Software Development Efficiency on the TMS320C6000 VelociTI Architecture

Because the C6000 is highly parallel and flexible, the task of scheduling code in an efficient way is best completed by the TI code generation tools (compiler and assembly optimizer) and not by hand. This results in a convenient C framework to maintain code development on current products as well as reuse the same code on future products (C code, C with TI C6000 Language Extensions, and the Linear Assembly source can all be reused on future TI C6000 DSPs).

The final territory of programming is the hand-scheduled assembly language. At this level, the programmer is literally scheduling the assembly code to the DSP pipeline. Depending on the programmer's ability, she may achieve results similar to those achieved by the C6000 tools; however, she risks “man-made” pipeline scheduling errors that can cause functional errors and performance degradation. In addition, scheduled code may not be reusable on future C6000 family members, unlike the three levels of C6000 source code. Consequently, avoid the hand-coded assembly level “Limited Reuse” programming if at all possible.

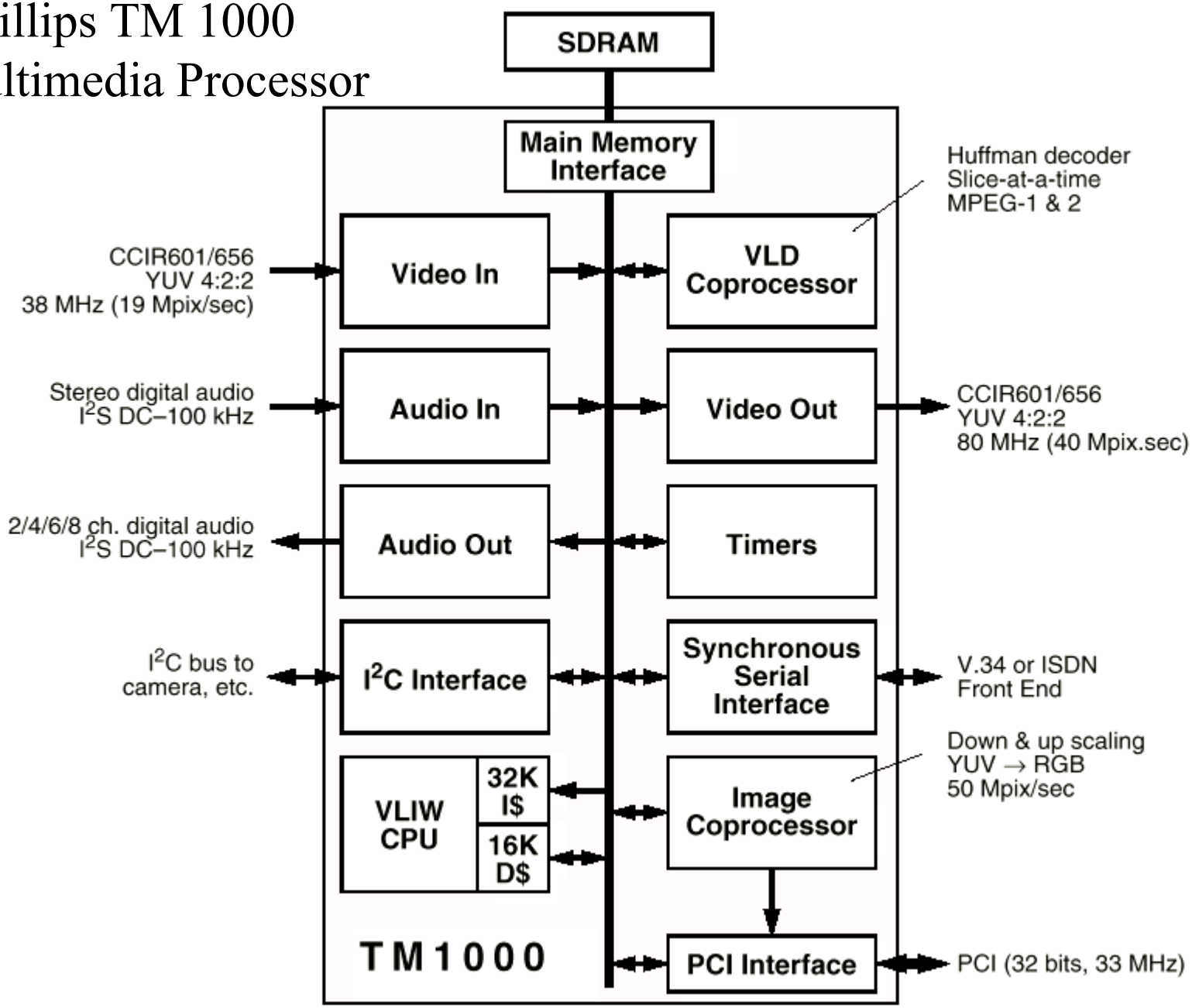
Table 8–3. TMS320C6x C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device†
uint **_lmbd(**uint s*rc1,* uint *src2***):**	**LMBD**	Searches for a leftmost 1 or 0 of *src2* determined by the LSB of *src1*. Returns the number of bits up to the bit change.	
int **_mpy(**int *src1,* int *src2***);** int **_mpyus(**uint *src1,* int *src2***);** int **_mpysu(**int *src1,* uint *src2***);** uint **_mpyu(**uint *src1,* uint *src2***);**	**MPY** **MPYUS** **MPYSU** **MPYU**	Multiplies the 16 LSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpyh(**int *src1,* int *src2***);** int **_mpyhus(**uint *src1,* int *src2***);** int **_mpyhsu(**int *src1,* uint *src2***);** uint **_mpyhu(**uint *src1,* uint *src2***);**	**MPYH** **MPYHUS** **MPYHSU** **MPYHU**	Multiplies the 16 MSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpyhl(**int *src1,* int *src2***);** int **_mpyhuls(**uint *src1,* int *src2***);** int **_mpyhslu(**int *src1,* uint *src2***);** uint **_mpyhlu(**uint *src1,* uint *src2***);**	**MPYHL** **MPYHULS** **MPYHSLU** **MPYHLU**	Multiplies the 16 MSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpylh(**int *src1,* int *src2***);** int **_mpyluhs(**uint *src1,* int *src2***);** int **_mpylshu(**int *src1,* uint *src2***);** uint **_mpylhu(**uint *src1,* uint *src2***);**	**MPYLH** **MPYLUHS** **MPYLSHU** **MPYLHU**	Multiplies the 16 LSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	

Table 8–3. TMS320C6x C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device†
int **_sadd(**int *src1,* int *src2***);** long **_lsadd(**int *src1,* long *src2***):**	**SADD**	Adds src1 to src2 and saturates the result. Returns the result.	
int **_sat(**long *src2***);**	**SAT**	Converts a 40-bit long to a 32-bit signed int and saturates if necessary.	
uint **_set(**uint *src2,* uint *csta*, uint *cstb***);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by csta and cstb, respectively.	
unit **_setr**(unit src2, int src1);	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by the lower ten bits of src1.	
int **_smpy(**int *src1*, int *sr2***);** int **_smpyh(**int *src1*, int *sr2***);** int **_smpyhl(**int *src1*, int *sr2***);** int **_smpylh(**int *src1*, int *sr2***);**	**SMPY** **SMPYH** **SMPYHL** **SMPYLH**	Multiplies src1 by src2, left shifts the result by one, and returns the result. If the result is 0x8000 0000, saturates the result to 0x7FFF FFFF.	
uint **_sshl(**uint *src2*, uint *src1***);**	**SSHL**	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result.	

Phillips TM 1000 Multimedia Processor



Trimedia TM-1000 (cont)

- Pipeline Basic Stages => Fetch, Decompression, Register-Read, Execute, Write-Back
- All instruction types share fetch, decompress, reg.read

⇒Alu, Shift, Fcomp

→Exe, Reg.Write

⇒DSPAlu

→Exe1, Exe2, Write

⇒ FPAlu, FPMul

→Exe1, Exe2, Exe3, Write

⇒Load/Store

→Address Compute, Data Fetch, Align/Sign Ext or Store update, write

⇒Jump

→Jmp Addr compute and condition check, fetch

Trimedia TM-1000

- Multimedia processor with a VLIW CPU core
- Five Execution Units, each EU is multi-function
  - ⇒27 functions total
  - ⇒Five Execution Units => Five operations per clock issued
- 15 Read and 5 Write Ports on register File
  - ⇒Need 15 read ports for 5 Execution Units because each operation requires two operands and a guard operand.
  - ⇒Guard operand makes each operation conditional based upon value of LSB of the guard operand
    - →Guard operand reduces number of branches needed, helps fill up branch delay slots.
  - ⇒128 Registers (r0, r1 always 0)

Trimedia TM-1000 (cont)

⇒Data fetched one cache line at a time (64 bytes), concatenated with left over data from previous fetch

⇒Multiple instruction sizes

→ 2 bits for NOP, 26 bits, 34 bits, and 44 bits.

→The current instruction actually has some bit information about the next instruction in order to simply decompression.

Table 4-1. Custom Operations Listed by Function Type

Function	Custom Op	Description
DSP absolute value	dspiabs	Clipped signed 32-bit absolute value
	dspidualabs	Dual clipped absolute values of signed 16-bit halfwords
DSP add	dspiadd	Clipped signed 32-bit add
	dspuadd	Clipped unsigned 32-bit add
	dspidualadd	Dual clipped add of signed 16-bit halfwords
	dspuquadaddui	Quad clipped add of unsigned/ signed bytes
DSP multiply	dspimul	Clipped signed 32-bit multiply
	dspumul	Clipped unsigned 32-bit multiply
	dspidualmul	Dual clipped multiply of signed 16-bit halfwords
DSP subtract	dspisub	Clipped signed 32-bit subtract
	dspusub	Clipped unsigned 32-bit subtract
	dspidualsub	Dual clipped subtract of signed 16-bit halfwords

More Operations

Sum of products	ifir16	Signed sum of products of signed 16-bit halfwords
	ifir8ii	Signed sum of products of signed bytes
	ifir8iu	Signed sum of products of signed/unsigned bytes
	ufir16	Unsigned sum of products of unsigned 16-bit halfwords
	ufir8uu	Unsigned sum of products of unsigned bytes
Merge, pack	mergelsb	Merge least-significant bytes
	mergemsb	Merge most-significant bytes
	pack16lsb	Pack least-significant 16-bit halfwords
	pack16msb	Pack most-significant 16-bit halfwords
	packbytes	Pack least-significant bytes
Byte averages	quadavg	Unsigned byte-wise quad average
Byte multiplies	quadumulmsb	Unsigned quad 8-bit multiply most significant
Motion estimation	ume8ii	Unsigned sum of absolute values of signed 8-bit differences
	ume8uu	Unsigned sum of absolute values of unsigned 8-bit differences