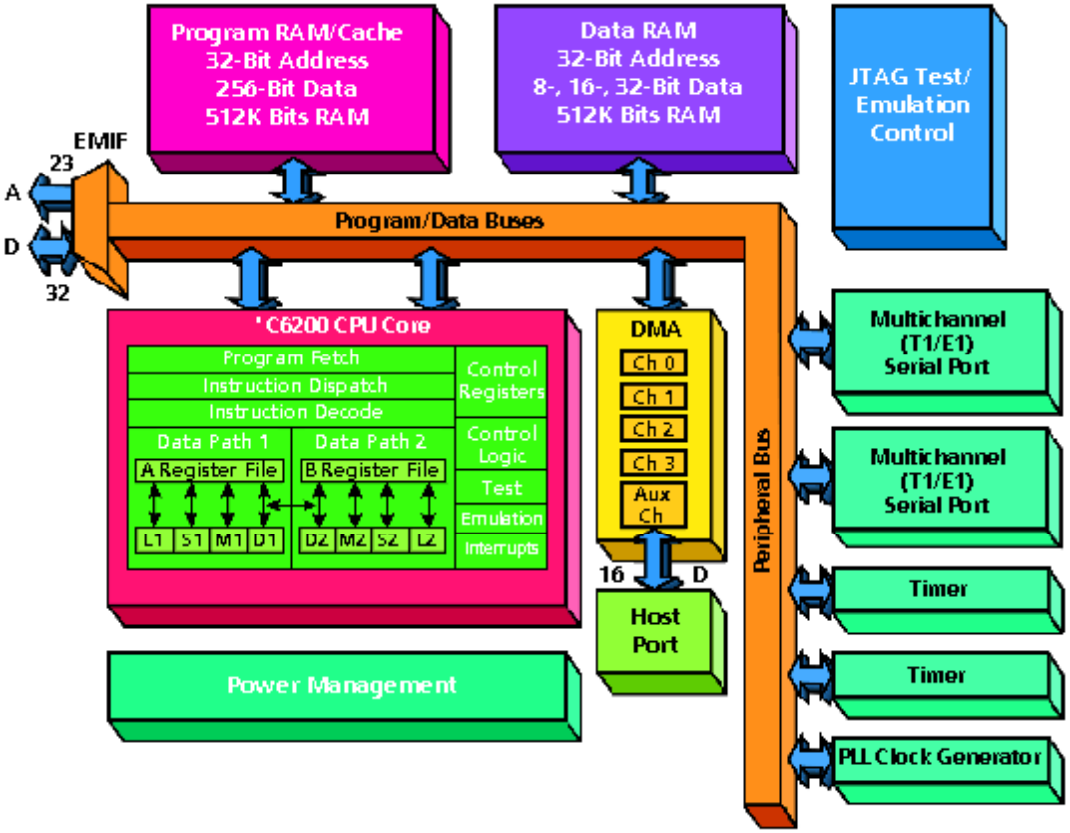
The VelociTI Architecture of the TMS320C6x DSP

TI slide + some addition

2hr presentation time

School of Electrical Engineering
Seoul National University

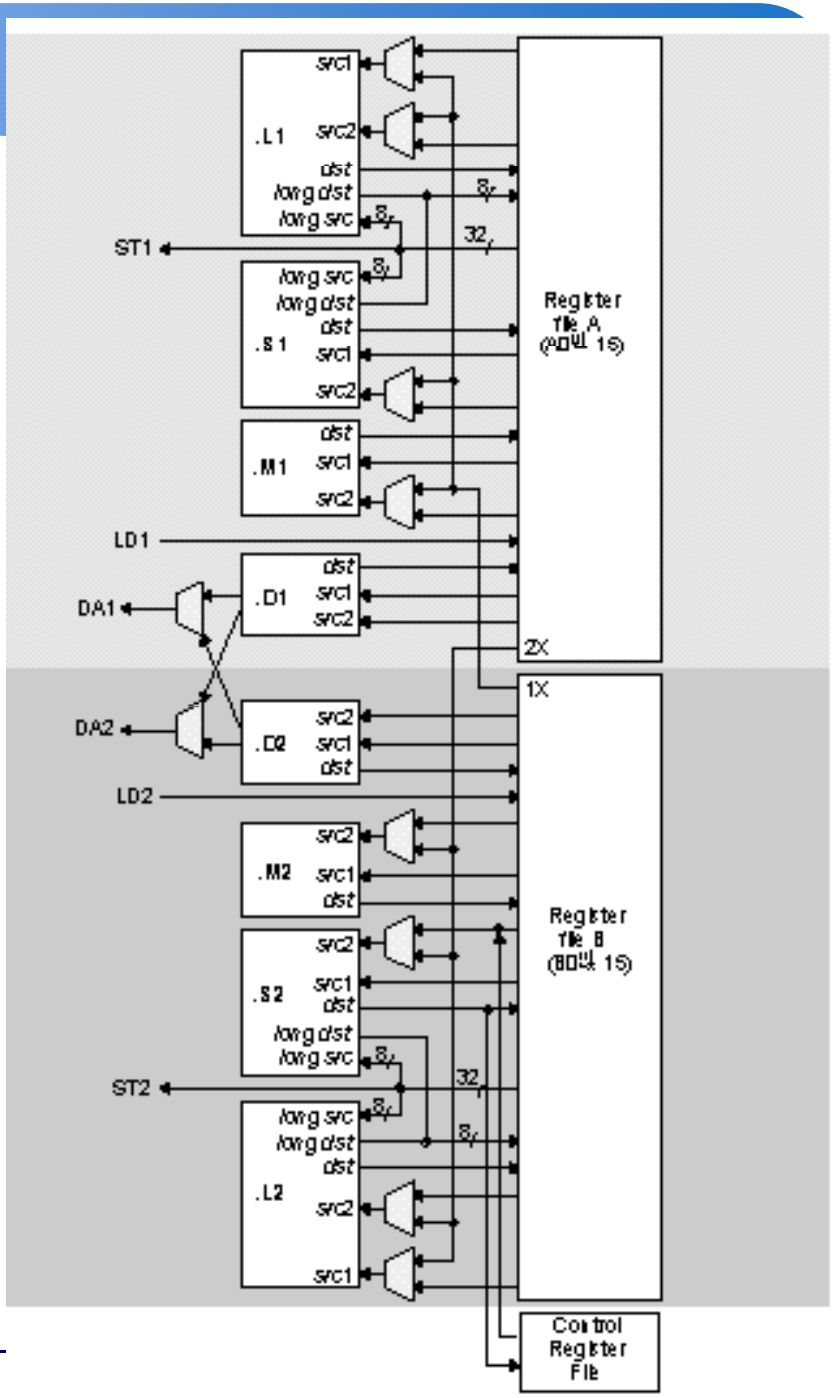
TMS320c6201 Architecture



- 1600 MIPS@200 MHz -> 1GHz
- 5 ns cycle time -> 1ns
- Up to 8 32-bit inst./cycle
- 3.3V I/O, 2.5V internal
- 0.25 micron, 5-layer metal
- 1 Mbit on-chip RAM
- SRAM, SB-SRAM, SDRAM interface
- 4 channel DMA
- 2 multi-channel T1/E1 serial ports
- 16-bit DMA host port
- 352-pin BGA

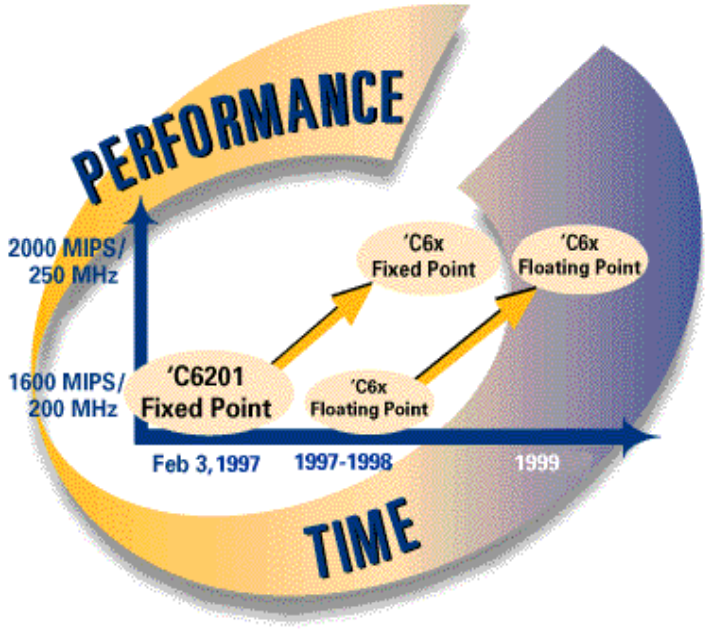
C62xx Datapaths

- **2 Datapaths**
- 8 Functional units
  - **orthogonal/independent**
  - **6 Arithmetic units**
  - **2 Multipliers**
- **Control**
  - Independent
  - Up to 8 32-bit inst. in parallel
- **Registers**
  - 2 Files
  - 32, 32-bit Registers Total
- **Cross paths (1X, 2X)**

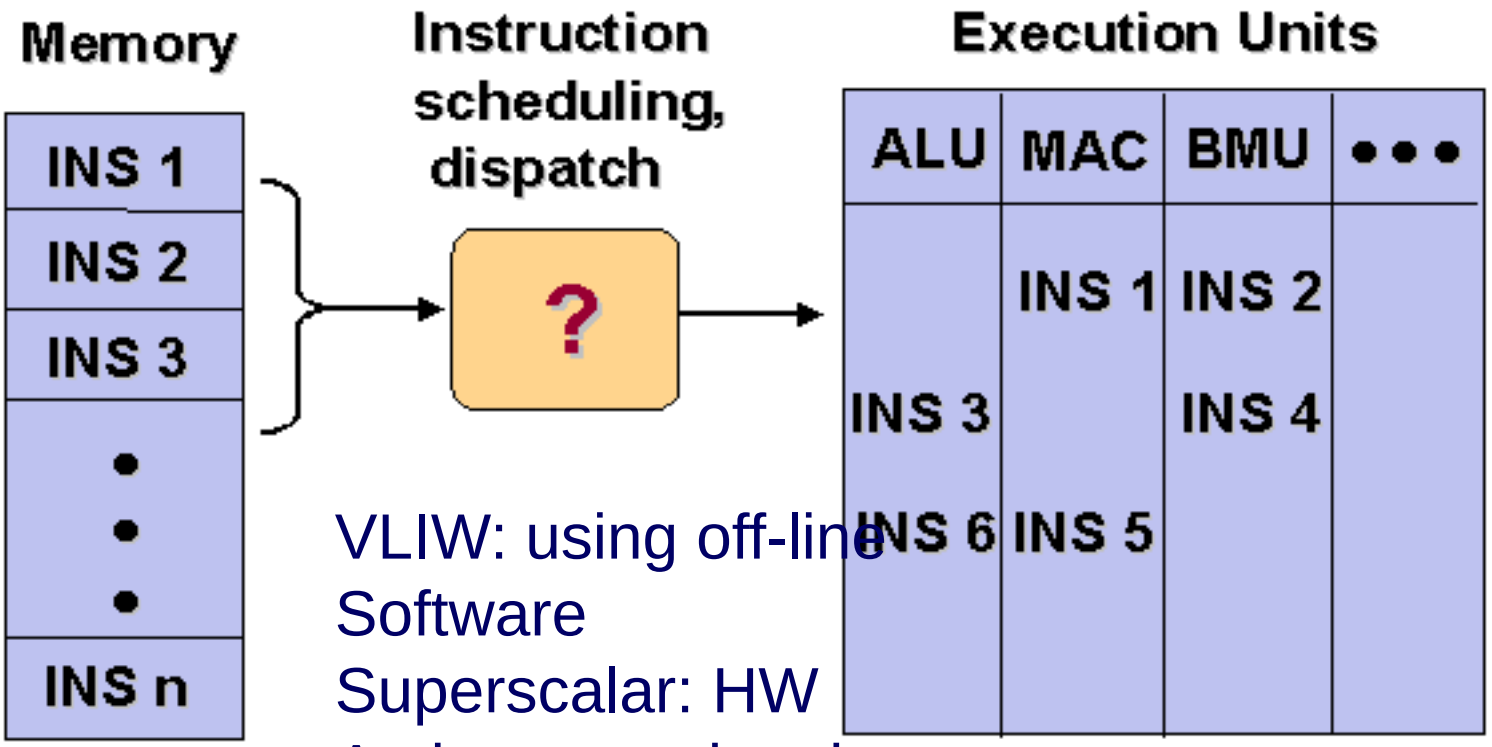


C62xx Product Objectives

- **High Performance**
  - Advanced VLIW CPU
  - **Max 8 instructions per cycle**.
  - 300 MHz ('C6203) -> 1GHz
  - Low Power/Performance (?)
- **Ease of Use**
  - Orthogonal RISC-like architecture
    - Low code density, no micro-parallelism with an instruction. (<-> traditional DSP)
  - Development Environment
    - **Efficient C compiler**
    - Assembly Optimizer (automatic parallelizer)
- **Newest semiconductor technology emplo**
  - **Low price even for small quantities**
  - **Large on-chip memory**
  - **Continuous update**



VLIW vs. Superscalar



VLIW: using off-line Software

Superscalar: HW

At the execution time

Superscalar vs VLIW

- **Superscalar:**
  - Scheduling at the execution time
  - Code scheduling scope is limited to a basic block
  - Complex HW scheduler – speed bottleneck
  - Code compatibility
- **VLIW**
  - Scheduling at the compile time (by SW)
  - Code scheduling scope is very wide
    - Virtually no scheduling boundary in a program
  - HW is simple (no scheduling operation)
  - No code compatibility (recompile needed)

Why VLIW (Very Long Instruction Word) ?

- **Superscalar disadvantages:**
  - Energy consumption is a major challenge
  - Dynamic behavior complicates software development
    - Execution-time variability can be a hazard
- **VLIW disadvantages:**
  - New kinds of programmer/compiler complexity
    - Programmer (or code-generation tool) must keep track of instruction scheduling
    - Deep pipeline, long latencies can be confusing, may make peak performance elusive
  - Code size bloat -> larger energy consumption
    - High program memory bandwidth requirements
- *VLIW lends well to DSP algorithms and offers possibilities for very high performance!*

Why VLIW?

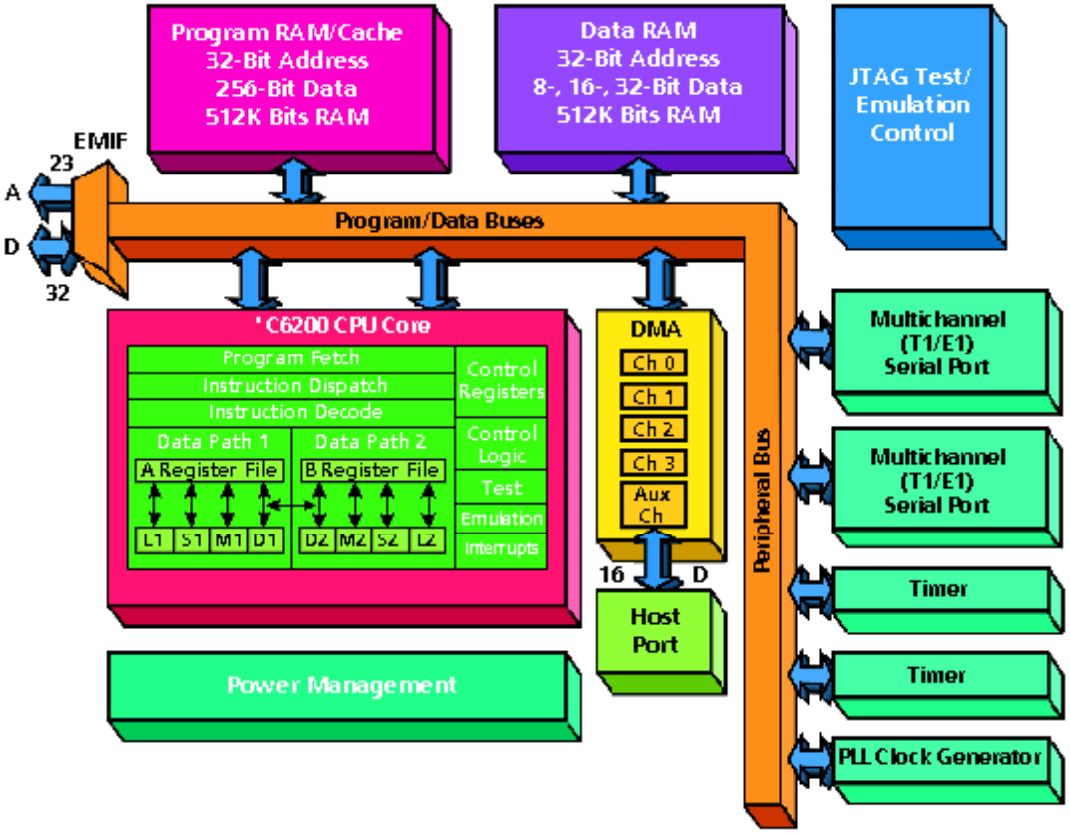
- **Characteristics:**
  - Multiple independent operations per cycle, packed into single large "instruction" or "packet"
  - More regular, orthogonal, RISC-like operations
  - Large, uniform register sets
  - ***Compiler-friendly***: orthogonal, deterministic, 100% conditional RISC-like instruction set
  - Advanced compiler and optimization technologies
    - Long history of VLIW compiler in the computer research area.
- **Examples of current & upcoming VLIW architectures for DSP applications:**
  - TI TMS320C6xxx, Siemens Carmel, ADI TigerSHARC

	RISC	Super Scalar	VLIW	Prog DSP	Hard-ware
DSP performance	low	high	Very high	medium	Very high
Hardware	simple	Very complex	complex	medium	Simple~complex
Application development efficiency (Compiler)	high	high	High (efficient compiler)	Medium (assembly, inefficient compiler)	Low (VHDL programming)
Code compa-tibility	Good	Good	Recompile needed (for embedded systems)	Good?	low
Clock frequency	high	Medium~low	high	Medium~low	

C62xx Targeted Applications

- **Multi-Channel: multiple channels of same application**
  - Cellular base-stations, Pooled modems, Central office switches, Multi-channel line echo cancellation, Multi-channel vocoders, Head end cable modem, Central office xDSL
- **Multi-Function: multiple applications**
  - Modem + Voice + Sound + ...
  - Pooled modem data pump + Control
  - Multimedia
- **Performance driven**
  - cable modem
  - xDSL
  - advanced terminals

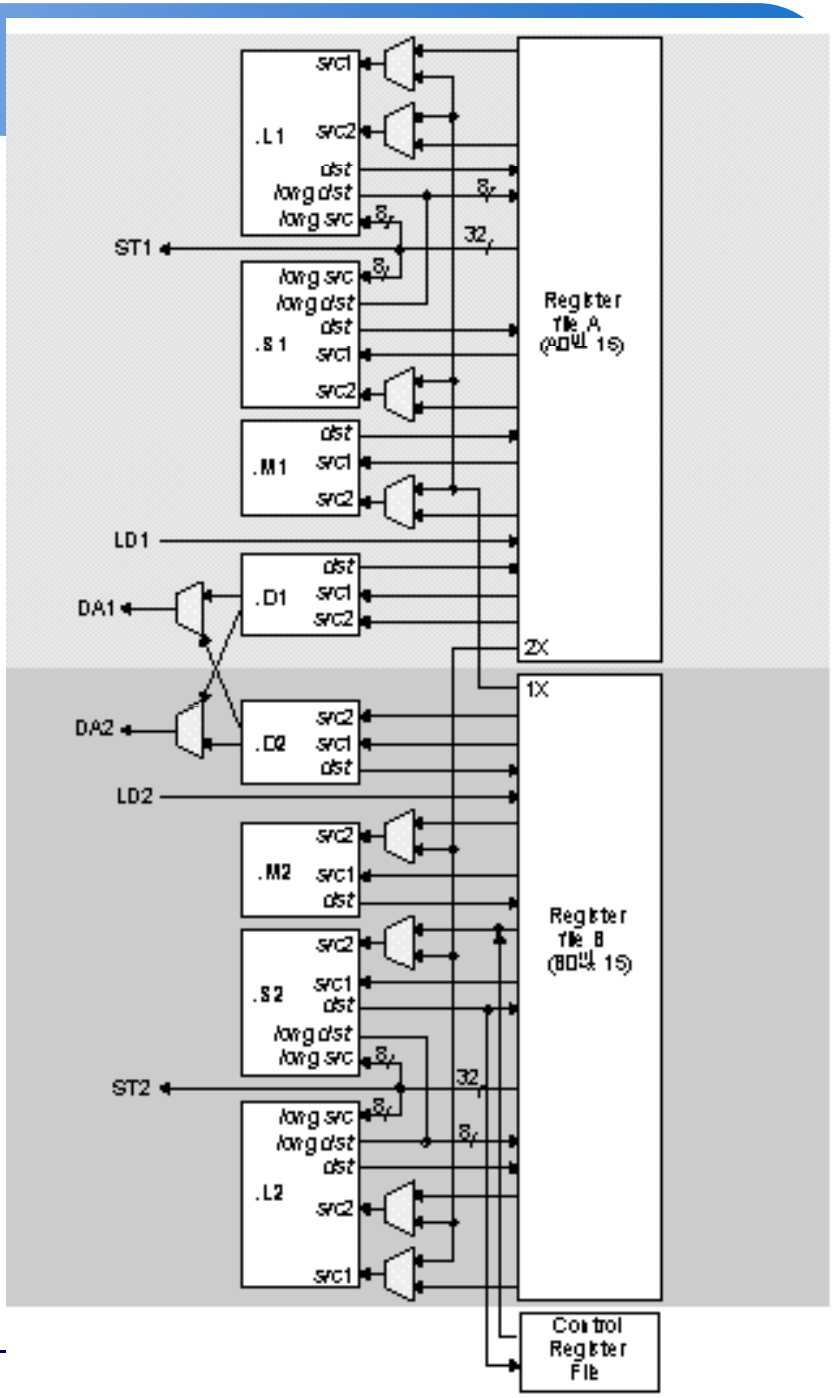
TMS320c6201 Architecture



- 1600 MIPS@200 MHz
- 5 ns cycle time
- Up to 8 32-bit inst./cycle
- 3.3V I/O, 2.5V internal
- 0.25 micron, 5-layer metal
- 1 Mbit on-chip RAM
- SRAM, SB-SRAM, SDRAM interface
- 4 channel DMA
- 2 multi-channel T1/E1 serial ports
- 16-bit DMA host port
- 352-pin BGA

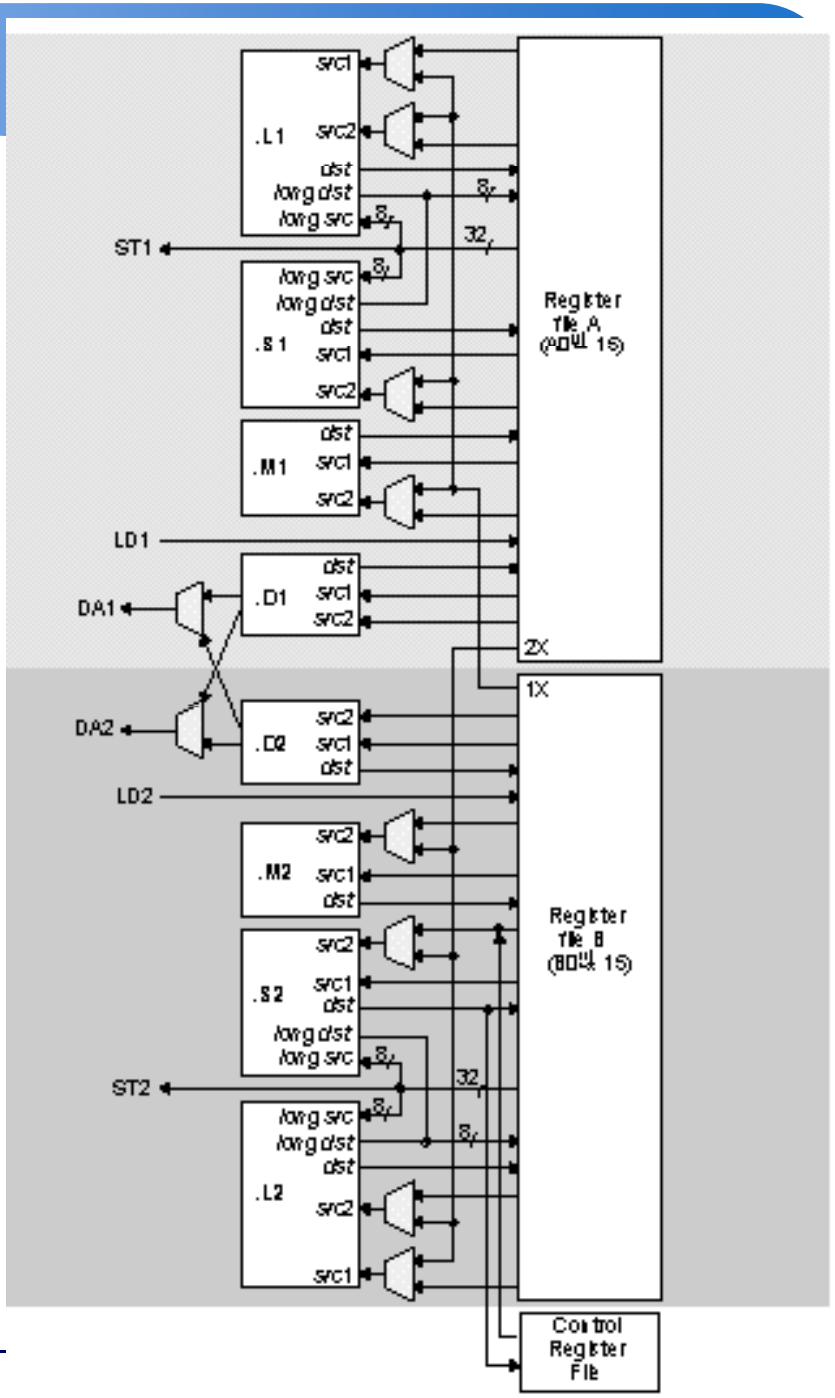
C62xx Datapaths

- **2 Datapaths**
- 8 Functional units
  - **orthogonal/independent**
  - **6 Arithmetic units**
  - **2 Multipliers**
- **Control**
  - Independent
  - Up to 8 32-bit inst. in parallel
- **Registers**
  - 2 Files (why two, not one or four?)
  - 32, 32-bit Registers Total
- **Cross paths (1X, 2X)**



C62xx Datapaths

- **L-unit (L1, L2)**
  - 40-bit integer ALU
  - Comparisons
  - Bit counting
  - Normalization
- **S-unit (S1, S2)**
  - 32-bit ALU
  - 40-bit shifter
  - Bitfield operations
  - Branching
- **M-unit (M1, M2)**
  - 16 x 16 -> 32
- **D-unit (D1, D2)**
  - 32-bit add/subtract
  - Address calculation



Weakness of the architecture

- **High instruction bandwidth**
  - Max 32bit*8 = Max 256 bit/cycle
  - L1, L2 cache/memory
  - There are researches on code compression of VLIW CPU
- **Low code density**
  - The instruction set is RISC style, no application specific, powerful, instructions
  - General purpose register based
  - This seems for good compiler.

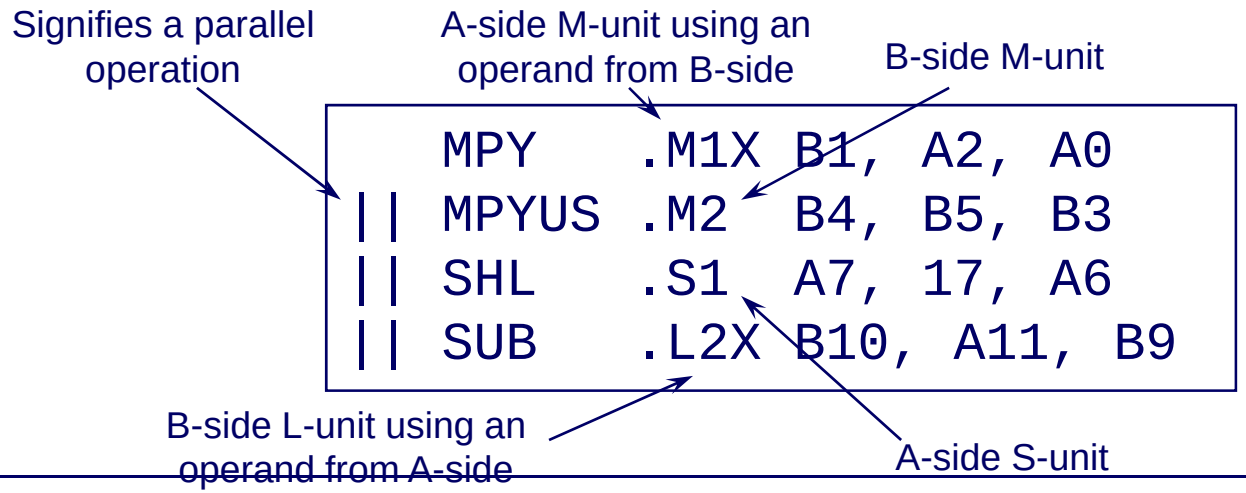
C62xx Instruction Set Features

Parallel Instructions

- Up to 8 instructions executed in parallel
- Determined at assembly or compile time

```
A0 = B1 * A2;
B3 = (unsigned) B4 * (signed) B5;
A6 = A7 << 17;
B9 = B10 - A11;
```

<- left operations are all independent



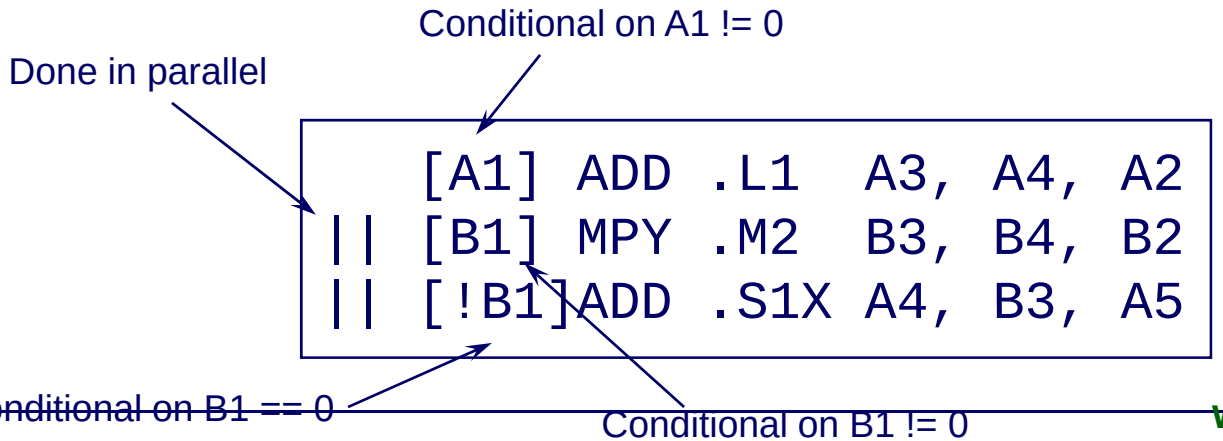
C62xx Instruction Set Features

Conditional Instructions

- **All Instructions can be conditional (predicate instructions)**
  - A1, A2, B0, B1, B2 can be used as conditions
  - Based on Zero or Non-Zero value
  - Compare instructions can allow other conditions (<, >, etc.)
- **Reduces branching**
- **Increases parallelism**

```
if (A1) A2 = A3 + A4;
if (B1) B2 = B3 * B4;
else A5 = A4 + B3;
```

Note: branches are Very expensive in deeply pipelined arch.



C62xx Instruction Set Features

Addressing

- **Load-Store architecture**
- **2 addressing units (D1, D2)**
- **Orthogonal: Any register can be used for addressing or indexing**
- **Signed/Unsigned byte, half-word, word addressable**
  - Indexes are scaled by type
- **Register or 5-bit unsigned constant index**
- **Indirect addressing modes**
  - Pre-Increment *++R[index], Post-Increment *R++[index]
  - Pre-Decrement *--R[index], Post-Decrement *R--[index]
  - Positive Offset *+R[index], Negative Offset *-R[index]
- **15-bit positive/negative constant offset from B14 or B15**
- **Circular addressing**
  - Fast and low cost: Power of 2 sizes and alignment
  - Up to 8 different pointers/buffers
  - Up to 2 different buffer sizes
- **Dual Endian Support**

C6201 Internal Memory Architecture

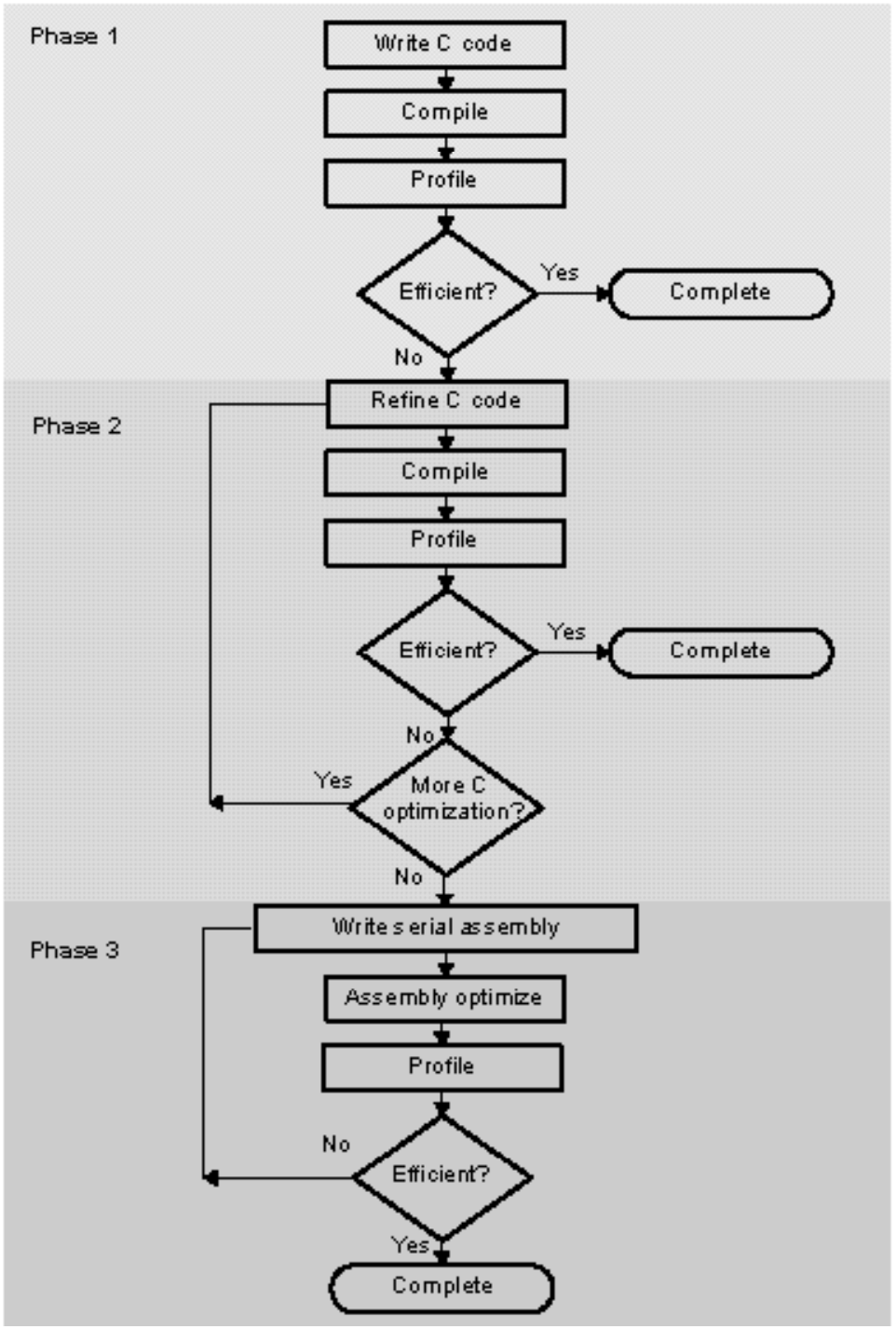
- **Separate internal program and data spaces**
- **Program**
  - 16K x **32-bit** instructions (2K fetch packets)
  - 256-bit fetch width
  - configurable as either
    - directed mapped cache
    - memory mapped program memory
- **Data**
  - 32K x **16-bit**
  - single ported accessible by both CPU data buses
  - 8 x 2K 16-bit banks
    - 2 memory spaces (4 banks each)
    - 4-way interleave
    - spaces and interleave minimize bank conflicts

C6201 Memory/Peripherals

- 4 channel DMA **w/ bootloading capability**
- 32-bit external memory interface **supporting**
  - 26-bit external byte address space
  - 32-bit width with byte-strobes
  - asynchronous & synchronous SRAM
  - synchronous DRAM
  - 8-bit/16-bit external ROM
- **16-bit host access port**
- **32K x 16 data memory**
- **16K x 32 program memory / instruction cache**
- **Peripherals**
  - 2 timers
  - 2 enhanced buffered T1/E1 serial ports

Code Generation Flow

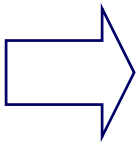
- *Compiler-friendly:*
  - start with C-level coding
  - optionally hand optimize only most critical functions
- **Tool suit support optimizing:**
  - Use ***C compiler*** to optimize and software pipeline
  - Use ***Assembly Optimizer*** to automatically schedule and optimize serial assembly code
  - Debug code through intuitive Windows-based source code (C and assembly) debugger



Using Intrinsics

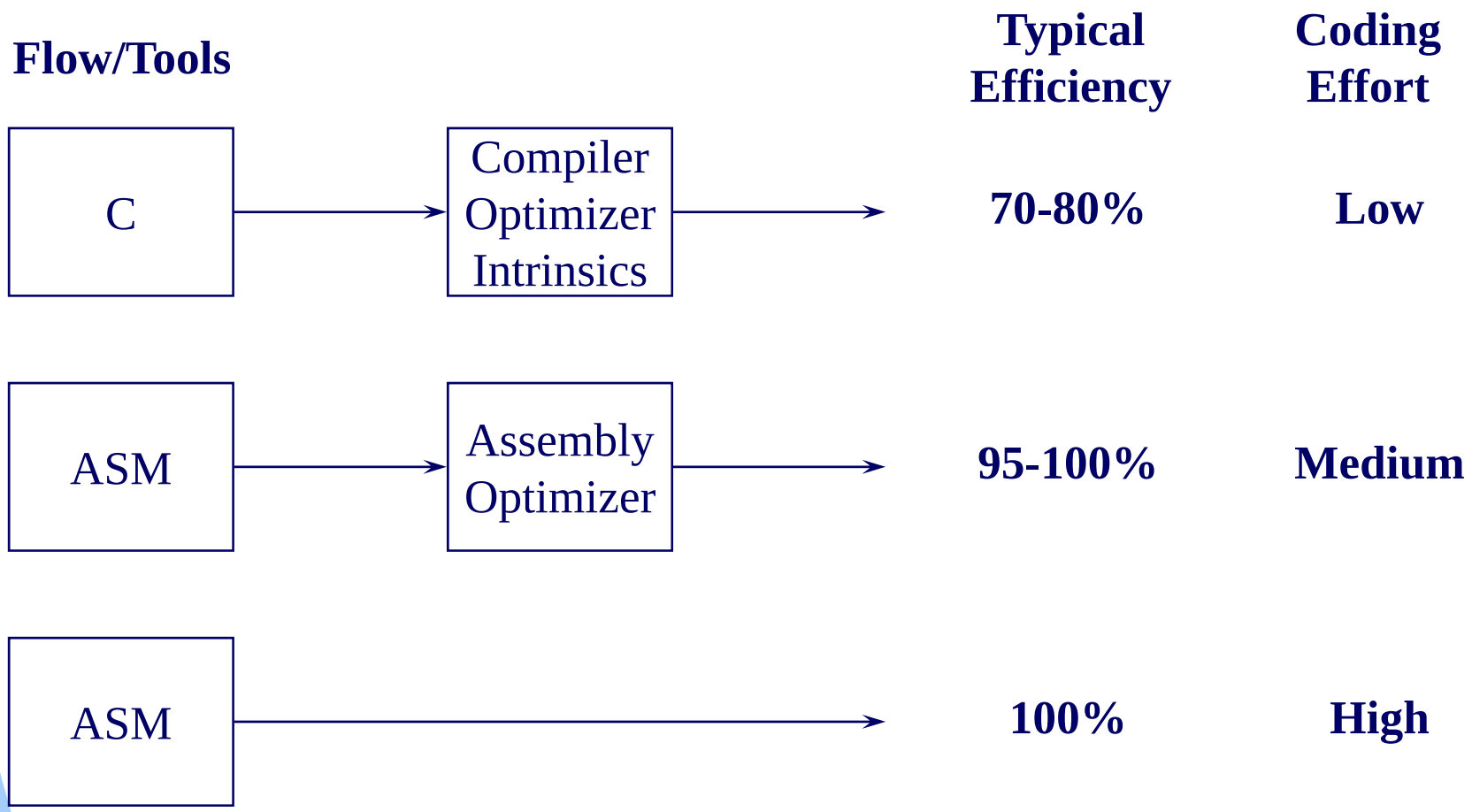
- *Intrinsic:*
  - Special function that maps directly to inlined C programs
- **e.g.**
  - int _add2(int src1, int src2); /* 16 x 2 Add */
  - int _sadd(int src1, int src2); /* Saturated Add */
  - int _sat(int src2); /* Saturation */
  - _mpyhl, _mpyhuls, _mpyhslu, _mpyluhs...

```
Int _sadd(int a, int b) /* saturated add */
{
    int result = a+b;
    if(((a^b) & 0x80000000)==0)
    {
        if((result^a) & 0x80000000)
            result = (a<0)? 0x80000000:0x7fffffff;
    }
    return result;
}
```



_sadd(a,b)

Code Generation Flow



Optimizing Assembly Code

- **Dot-Product C Code**

```
int dotp(short a[], short b[])
{
      int sum, i;
      sum = 0;
      for(i = 0; i<100; i++)
            sum += a[i] * b[i];
      return sum;
}
```

Serial Assembly

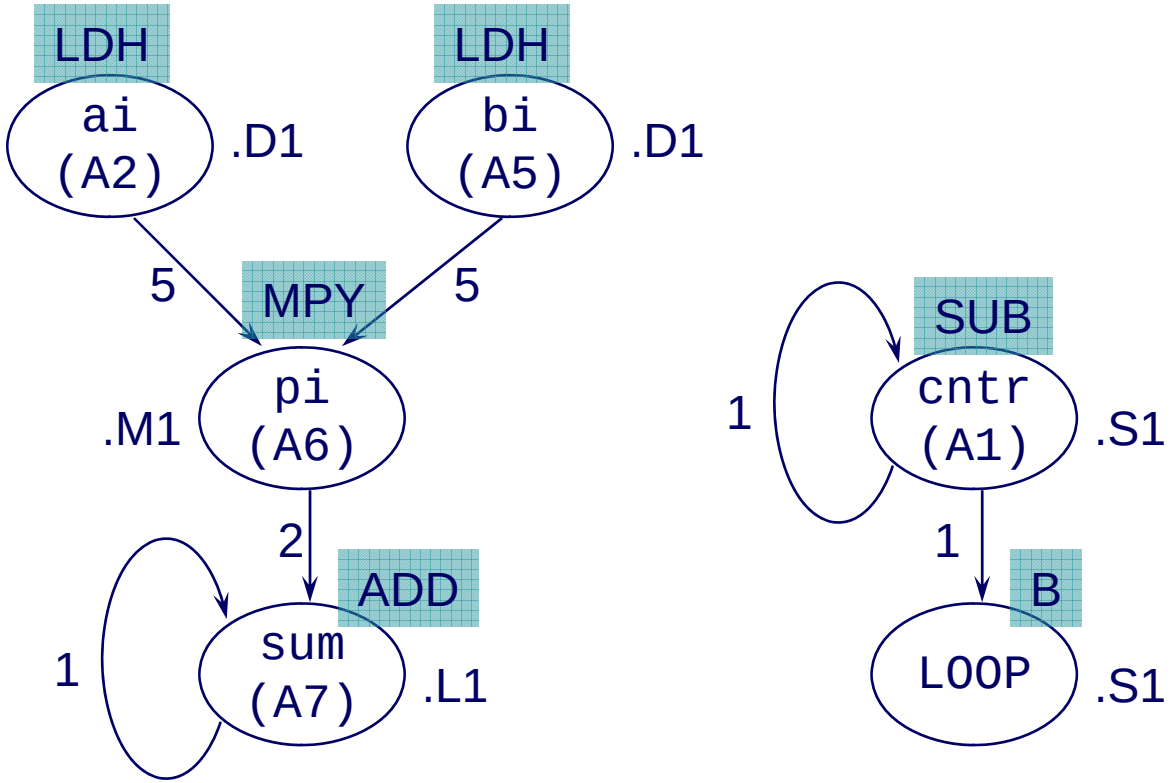
Dot-Product Serial Assembly

```
        MVK     .S1     100,A1   ;set up loop counter
        ZERO    .L1     A7       ;zero out accumulator
LOOP:
        LDH     .D1     *A4++,A2 ;load ai from memory
        LDH     .D1     *A3++,A5 ;load bi from memory
        NOP     4                ;delay slots for LDH
        MPY     .M1     A2,A5,A6 ;ai * bi
        NOP                      ;delay slot for MPY
        ADD     .L1     A6,A7,A7 ;sum += (ai*bi)
        SUB     .S1     A1,1,A1  ;decrement loop counter
[A1]    B       .S2     LOOP     ;branch to loop
        NOP     5                ;delay slots for branch
; Branch occurs here
```

- 100 Iterations: 2+100x16 = 1602

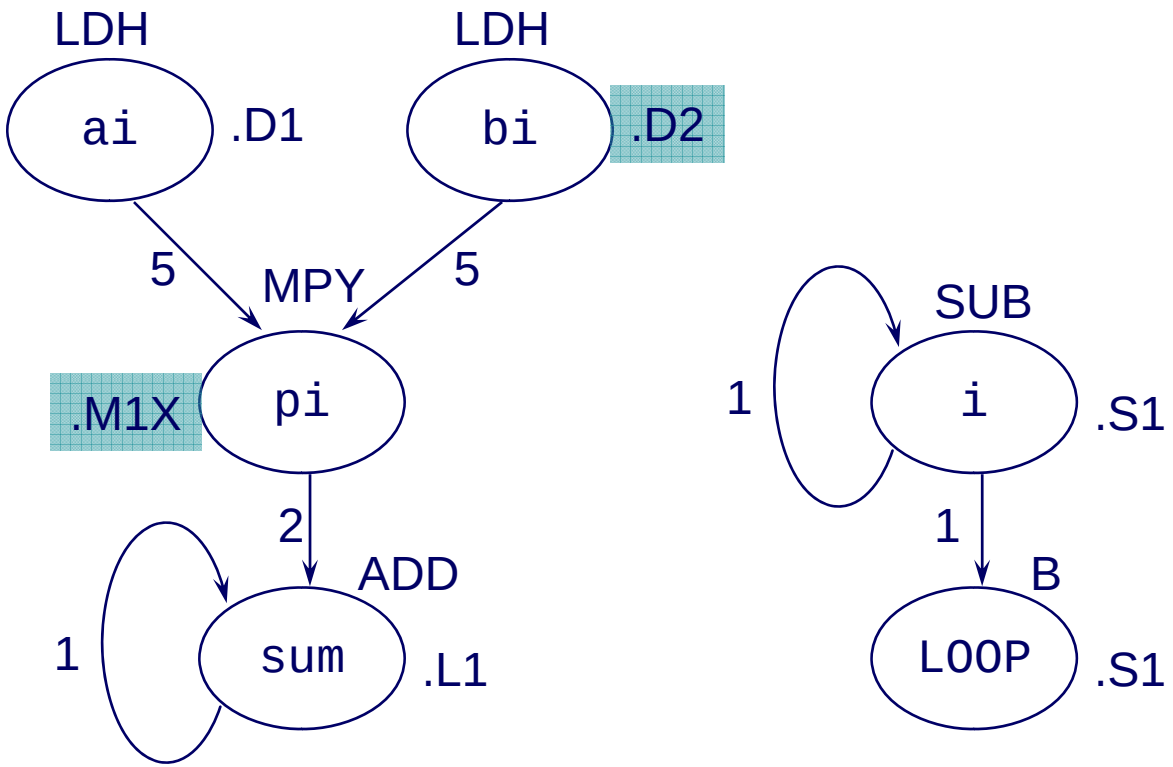
Serial Assembly

- **Dependency Graph for Dot-Product**



Parallel Assembly

❖ Dependency Graph for Parallel Assembly



Parallel Assembly

❖ Dot-Product Parallel Assembly

```
        MVK     .S1     100,A1   ;set up loop counter
||      ZERO    .L1     A7       ;zero out accumulator
LOOP:
        LDH     .D1     *A4++,A2 ;load ai from memory
||      LDH     .D2     *B4++,B2 ;load bi from memory
        SUB     .S1     A1,1,A1  ;decrement loop counter
[A1]    B       .S2     LOOP     ;branch to loop
        NOP     2                ;delay slots for LDH
        MPY     .M1X    A2,B2,A6 ;ai * bi
        NOP                      ;delay slot for MPY
        ADD     .L1     A6,A7,A7 ;sum += (ai*bi)
; Branch occurs here
```

❖ 100 Iterations: 1+100x8 = 801

Unrolled Loop

❖ Unrolled Dot-Product C Code

```
int dotp(short a[], short b[])
{
        int sum0, sum1, sum, i;
        sum0 = 0;
        sum1 = 0;
        for(i = 0; i<100; i+=2){
                sum0 += a[i] * b[i];
                sum1 += a[i+1] * b[i+1];
        }
        sum = sum0 + sum1;
        return sum;
}
```

Unrolled Loop Assembly with LDW

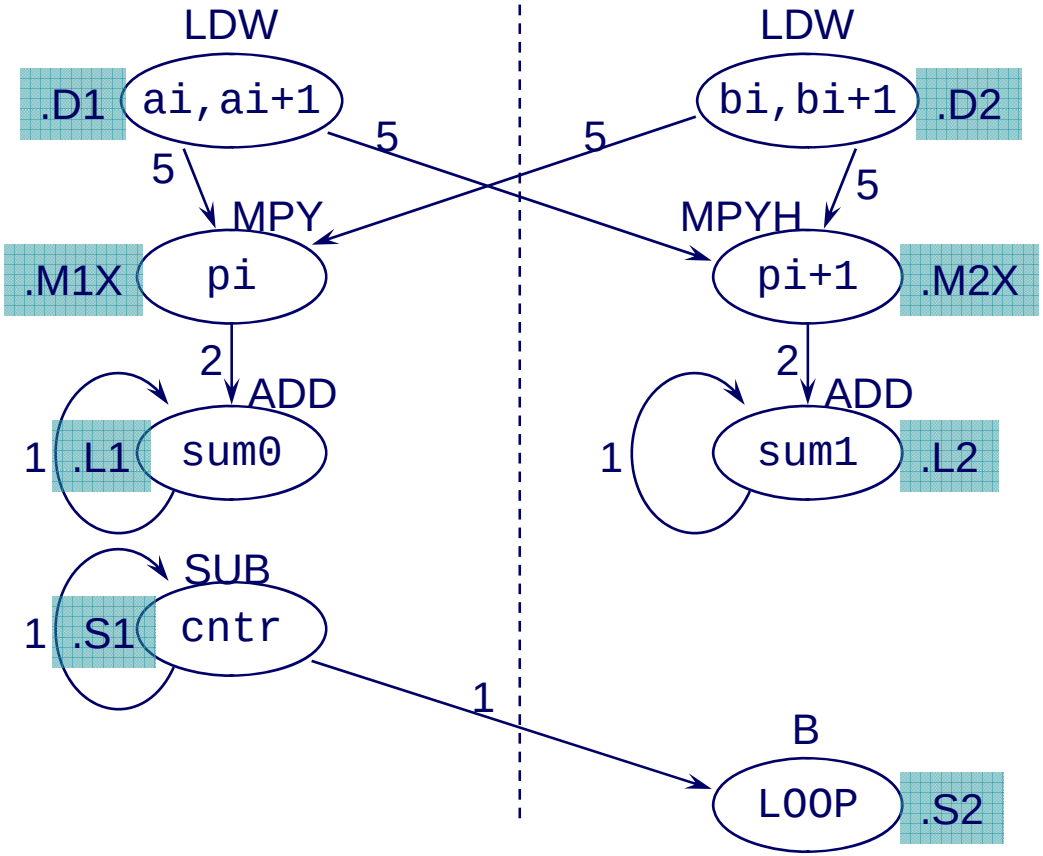
❖ Dot-Product Assembly with LDW

```
          MVK     .S1     50,A1    ;set up loop counter
||        ZERO    .L1     A7       ;zero out sum0 accumulator
||        ZERO    .L2     B7       ;zero out sum1 accumulator
LOOP:
          LDW     .D1     *A4++,A2 ;load ai & ai+1 from memory
||        LDW     .D2     *B4++,B2 ;load bi & bi+1 from memory
          SUB     .S1     A1,1,A1  ;decrement loop counter
[A1]      B       .S1     LOOP     ;branch to loop
          NOP     2
          MPY     .M1X    A2,B2,A6 ;ai * bi
||        MPYH    .M2X    A2,B2,B6 ;(ai+1) * (bi+1)
          NOP
          ADD     .L1     A6,A7,A7 ;sum0 += (ai*bi)
||        ADD     .L2     B6,B7,B7 ;sum1 += ((ai+1) * (bi+1))
; Branch occurs here
          ADD     .L1X    A7,B7,A4 ; sum = sum0 + sum1
```

❖ 100 Iterations: 1+50x8+1 = 402

Unrolled Loop Assembly with LDW

❖ **Dependency Graph for Dot-Product with LDW**



Software Pipelined Assembly

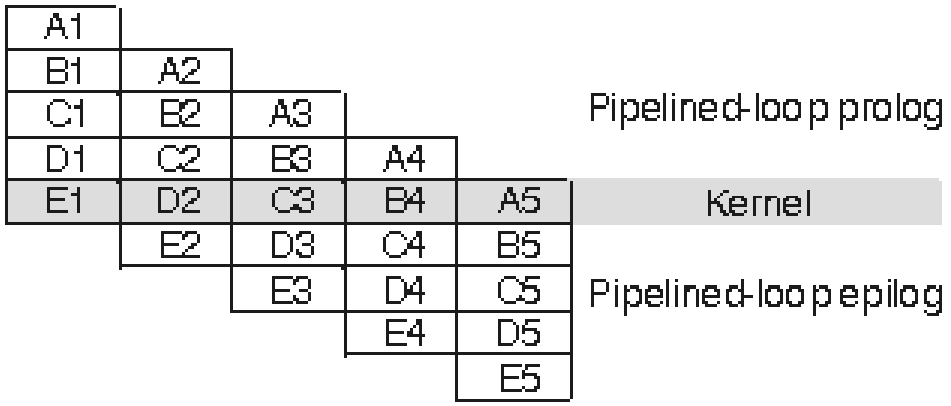
❖ Dot-Product Modulo Iteration Interval Table

Unit/Cycle	0,8,...	1,9,...	2,10,...	3,11,...	4,12,...	5,13,...	6,14,...	7,15,...
.D1	LDW							
.D2	LDW							
.M1						MPY		
.M2						MPYH		
.L1								ADD
.L2								ADD
.S1		SUB						
.S2			B					

**Resource만으로 볼 때 1 iteration/cycle 가능

Software Pipelining

- *Software pipelining* **is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel.**
- **What is pipelining? Supplying the input before the processing of the previous input is not completed.**



Software Pipelined Assembly

- **Dot-Product Modulo Iteration Interval Table**

Unit/Cycle	0	1	2	3	4	5	6	7	8
.D1	LDW (0)	LDW (1)	LDW (2)	LDW (3)	LDW (4)	LDW (5)	LDW (6)	LDW (7)	LDW (8)
.D2	LDW (0)	LDW (1)	LDW (2)	LDW (3)	LDW (4)	LDW (5)	LDW (6)	LDW (7)	LDW (8)
.M1						MPY (0)	MPY (1)	MPY (2)	MPY (3)
.M2						MPYH (0)	MPYH (1)	MPYH (2)	MPYH (3)
.L1								ADD (0)	ADD (1)
.L2								ADD (0)	ADD (1)
.S1		SUB (0)	SUB (1)	SUB (2)	SUB (3)	SUB (4)	SUB (5)	SUB (6)	SUB (7)
.S2			B (0)	B (1)	B (2)	B (3)	B (4)	B (5)	B (6)

- **100 Iterations: 7+50+1 = 58**

Software Pipelined Assembly

여기에 * Prologue Code

```
        <pipelined-loop prolog snipped here>
LOOP:
        ADD     .L1     A6,A7,A7 ;sum0 += (ai * bi)
||      ADD     .L2     B6,B7,B7 ;sum1 += (ai+1 * bi+1)
||      MPY     .M1X    A2,B2,A6 ;ai * bi
||      MPYH    .M2X    A2,B2,A6 ;ai+1 * bi+1
||[A1]  SUB     .S1     A1,1,A1  ;decrement loop counter
||[A1]  B       .S2     LOOP     ;branch to loop
||      LDW     .D1     *A4++,A2 ;load ai & ai+1 from memory
||      LDW     .D2     *B4++,B2 ;load bi & bi+1 from memory
; Branch occurs here

        ADD     .L1X    A7,B7,A4 ;sum = sum0+sum1
```

여기에 *Epilogue Code

Comparing Performance

- Comparison of Dot-Product Code Examples

Example	100 Iterations	Cycle Count
Serial Assembly	2+100x16	1602
Parallel Assembly	1+100x8	801
Unrolled Loop Assembly with LDW	1+50x8+1	402
Software Pipelined Assembly	7+50+1	58

Disadvantages of Software Pipeline Code

- Contains prologue and epilogue codes.
- Loop size needs to be large to hide the effects of the prologue and epilogue codes.
- May need non-pipelined code for the short loop length.
- _nassert statement: give information so that non-pipelined code is generated or not. _nassert(N>=10); this means N>= 10.

Software pipeline code

- Instrinsic allowed, function call not allowed
- Conditional break (early exit) not allowed
- Loop must count down, and terminates at 0.
- When the code size is too large and, therefore, requires more than 32 registers. -> no pipeline possible.
- A register value is too long. -> no pipeline.

Wonyong Sung
Multimedia Systems Lab SNU

Loop Unrolling

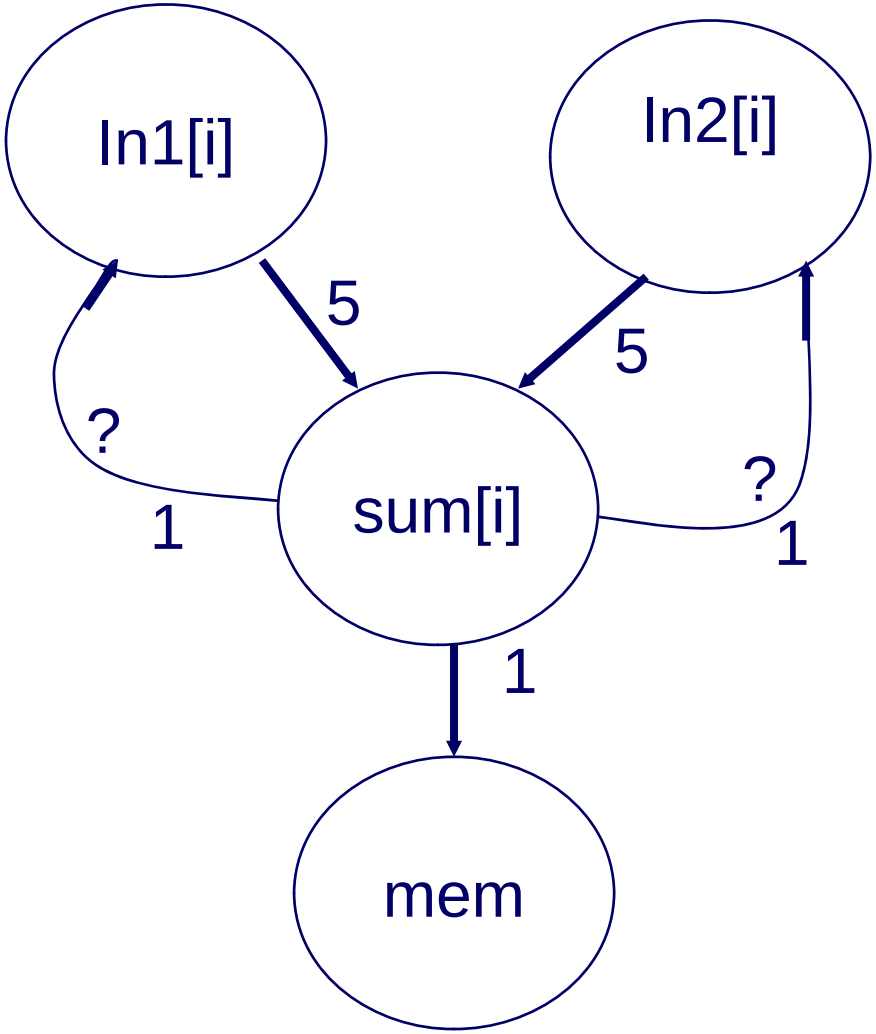
- Conventional processor: reducing the overhead of loop count (decrease and conditional jump).
- VLIW: devise the loop so that there exist enough number of instructions in the loop.
  - Ex: FIR filtering – one multiplication for each tap, which means only utilize half of the resource. So, it is needed to change the code so that two taps, or more, are processes at each iteration.

Basic Vector Sum

```
void vecsum(short *sum, short *in1, short *in2,
    unsigned int N)
{ int i;
for (i=0; i<N; i++)
  sum[i] = in1[i] + in2[i];
}
```

Condition for parallelization: sum does not affects in1, in2. (no dependency from sum to in1, in2)

Resource dependency problem.


In1[i]
In2[i]
5
5
?
1
sum[i]
?
1
1
mem

- **Solution: const keywords**

```
void vecsum(short *sum, const short *in1, const short
   *in2, unsigned int N)
{ int i;
for (i=0; i<N; i++)
  sum[i] = in1[i] + in2[i];
}
```

Development strategies

- Identify loops and time consuming portions.
- Reduce memory dependency, arithmetic dependency. It should be known to the compiler.
- Limitation due to resource
  - memory: two 32bit data/cycle
  - mul: two 16*16
  - ALU etc.
- Instrinsic, software pipelining, loop unrolling, count down loop
- Short loops do not get benefit from the software pipelining. Big array consumes the internal memory.

FIR filter

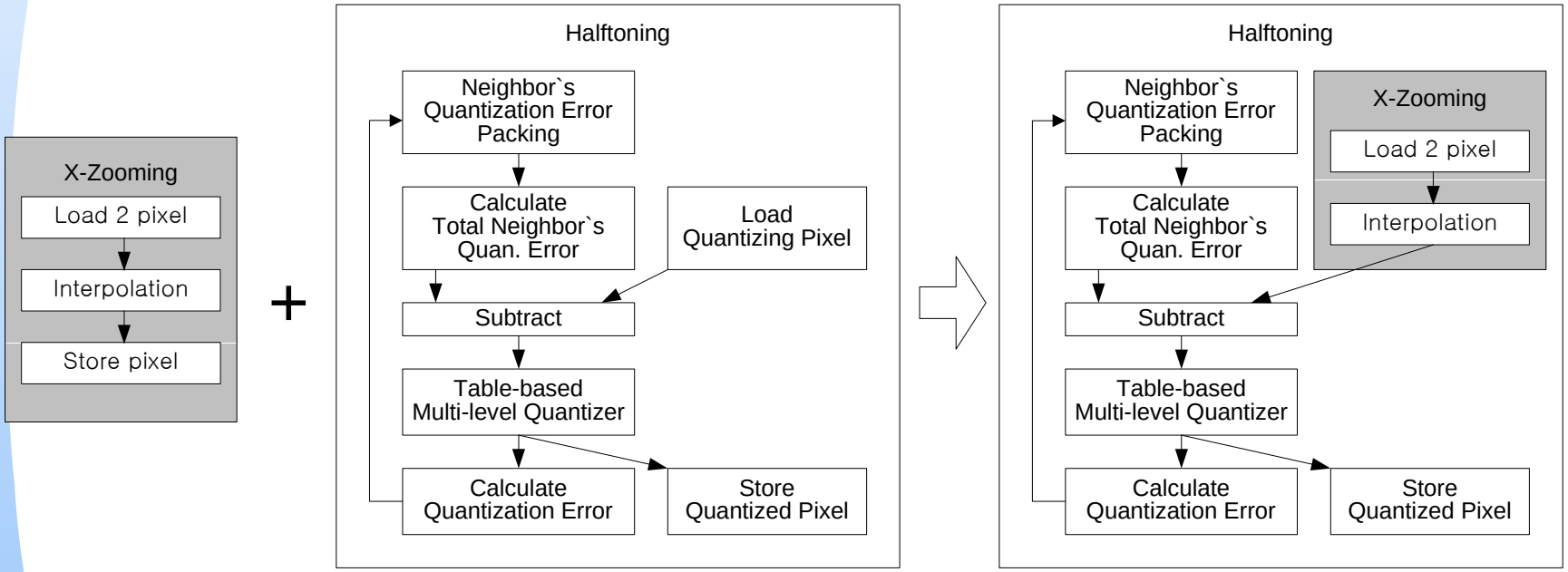
- Input data, coefficients should not overlap with the output storage.
- Conventional code: 1 tap/loop
  16 bit data,coef fetch, 1 multiplier, 1 add
  -> do not fully utilize the resources.
- Loop unrolling: 2 tap/loop
  two LDW (1 upper, 1 lower 16-bit), 2 multiplier, 2 add, 1 count sub and jump
- Intrinsic (mul, mulh), software pipelining (when tap length is not small.)

Load & store reduction

- **Why important?**
  - C6x has a RISC style instruction set (load store machine), memory access is expensive.
  - Aligned (16byte boundary) memory load: 128 bit supported, non-aligned memory load: 64 bit.
- **Usually load-store reduction can be conducted by**
  - loop fusion
  - function merging
  - multi-block processing

Example of function merging in the digital copier program

Fig. 3. Merging X-zooming and Vector error diffusion.



Example of multi-block processing in ME

- **Motion estimation: 4x4 SAD (Sum of Absolute Difference) computation intensive with non-aligned memory accesses inevitable**
  - Each data unit is just 1 byte (8bit) so SIMD computation is needed

Number of blocks for each loop	One block of 4x4	Two blocks of 4x4	Four blocks of 4x4
Non-aligned 4byte load	4	-	-
Aligned 4byte load	5	1	1
Non-aligned 8byte load	-	4	8
Aligned 8byte load	-	4	8
Aligned 4byte store	1	-	-
Aligned 8byte store	-	1	2
SUBABS4	4	8	16
DOTPU4	4	8	16
Parallelism of the loop kernel	5.67	5.33	6.07
Number of cycles per pixel for SAD computation	0.562	0.375	0.234

Conditional or branch

- **C64x is intended for repetitive execution of arithmetic intensive algorithms**
  - But what if not?
- **It's unavoidable to handle control intensive code**
- **Branch penalty is very big**
  - It needs to flush the pipeline
  - May have to wait until the conditions are known.
  - The penalty is proportional to the number of simultaneously executable instructions
- **C6x provision**
  - Conditional execution – program flow is linear (do not destroy the pipeline), just some instruction may or may not be executed according to the conditions
  - However the condition, and the conditional execution body need to be simple.

C language based development 1

- **C/C++ source file**
- **Parser: generate .if file**
- **Optimizer: generate .opt file**
  - Optimization levels (-O1 ~ -O3)
- **Code generator: generate .asm file**
  - Conduct processor specific optimizations

C program based developments

- **Optimization levels (-O0, -O1, ..)**
  - -O0: performs flow graph simplification
    - Allocate variables to registers
    - Performs loops rotation, eliminates unused code…
  - -O1: performs local copy/constant propagation
    - Eliminates local common expressions
  - -O2: performs software pipelining and loop opt.
    - Performs loop unrolling, eliminates global common subexpressions
  - -O3: Removes functions that are never called, inlines calls to small functions, identifies file-level variable characteristics
- **Program level optimizatin (-pm and –O3 options)**
  - All of the source files are compiled into one intermediate file called a module
    - If a function is not called directly or indirectly, the compiler removes the function
    - return value of a function is never used, the compiler deletes the return code

Software pipelining related issues in C programming

- **Turn off sw pipelining for dubugging..: -mu**
  - To reduce the code size: use –ms2, -ms3
- **Terms define in the SW pipelining information**
  - Loop unrolling factor: the factor that the loops is unrolled to increase the performance based on the resource bound constraint. Odd case
  - Known minimum (maximum) trip count: the number of times the loop was executed
  - Loop carried dependency bound: the distance of the largest loop carry path, one iteration writes a value that must be read in a future iteration. Marked with ^ symbol.
  - Iteration interval: the number of cycles between the initiation of successive iterations
  - Resource bound: the most used resource constrains the min iteration interval. Unpartitioned and partitioned (A and B)
  - Resource partition: .L, .S, .D, .M ….

SW pipelining with unknown trip counts

- **Too small trip count – sw useless**
  - The best is let it be known
- **Other techniques that compiler does**
  - Multi-version code generation
    - One sw pipelined and the other not
    - Check the trip count in the run time and determines which code to execute.
    - Increase the code size
  - Prolog and epilog collapsing – relieve the requirements of min trip count

```
loop: ins1
      ins2  ||     dec n    ; n = n-1
      ins3  || [n] br loop  ; branch to loop
                            ; if n>0
```

```
loop:   sub n,2,n

        ins1                                              ; prolog stage 1
        ins2 || ins1 || dec n                             ; prolog stage 2
        ;-----------------------------------------------------------------
kernel: ins3 || ins2 || ins1 || [n] dec n || [n] br kernel ; kernel
        ;-----------------------------------------------------------------
               ins3 || ins2                               ; epilog stage 1
                       ins3                               ; epilog stage 2
```

Figure 1: Software-pipelined Loop

```
loop:    sub n, 1, n                          ; exec. kernel n-2+1 times

         ins1                                 ; prolog stage 1
         ins2 || ins1 || dec n                ; prolog stage 2
         ;-----------------------------------------------------------------
kernel:  ins3 || ins2 || ins1 || [n] dec n || [n] br kernel
         ;-----------------------------------------------------------------
                 ins3                         ; epilog stage 2
```

Figure 2: Software-pipelined loop with one epilog stage collapsed.

```
loop:                                         ; exec. kernel n-2+2 times
         sub n, 1, p                          ; p = n - 1
         ;-----------------------------------------------------------------
         ins1                                 ; prolog stage 1
         ins2 || ins1 || dec n                ; prolog stage 2
         ;-----------------------------------------------------------------
kernel:  ins3 || [p] ins2 || ins1 || [p] dec p || [n] dec n || [n] br kernel
```

Figure 3: Software-pipelined loop with both epilog stages collapsed

n>= 3

It were safe
to execute ins1
extratime

Investigative feedback

- **Loop carried dependency bound is much larger than unpartitioned resource bound**
  - May be memory alias disambiguation needed
- **Two loops are generated one not sw pipelined: when the trip count can be too low. One is a non-pipe version**
- **Uneven resource: loop unrolling helps**
- **Larger outer loop overhead in nested loops: inner count is small -> loop unrolling of inner most loop**
- **Memory bank conflicts: two memory accesses are 32bytes apart on C64 and both accesses reside within the same memory block, a memory bank stall will occur.**

Compiler optimization techniques (1)

- **Cost based register allocation**
  - Variables used within loops are weighted to have priority over others, variables do not overlap can be allocated to the same reg.
- **Strength reduction**
  - Turns the array references into efficient pointer references with autoincrements
- **Alias disambiguation**
  - two or more pointer (or structure) references refer to the same memory location. In this case, this aliasing of memory locations prevents compiler from retaining values in registers

Compiler optimization techniques (2)

- **Branch optimizations and control flow simplification**
  - Compiler analyzes the branching behavior and rearranges the linear sequences (basic blocks) to remove branches
  - When the value of a condition is determined at the compile time, the compiler can delete a cond branch.
  - Simple control flow constructs are reduced to conditional instructions

Compiler optimization techniques (3)

- **Data flow optimization**
  - Copy propagation
  - Common subexpression elimination
- **Expression simplification**
  - A=(b+4)-(c+1) -> a=b-c+3
- **Loop invariant code motion**
- **…**

Advanced techniques for conditional or branch

- **Speculative execution**
  - When the branch probability is not equal, e.g. loop
    - Branch prediction is implemented in HW for some CPU's.
  - Execute assuming the higher probability path, and then (if the assumption is wrong) undo the job.

❖ **Example of run-length code part**

```
for(coeff_ctr=0 ; coeff_ctr<16; coeff_ctr++)
{
    ij = *zz ++;
    i = ij &0x3;    j = ij >>2;
    run++;    ilev=0;
    level = level_buf[i][j];
    if(level >1)
   {
    *coeff_cost += MAX_VALUE;
     level = sign2(level,img->m7[i][j]);
     ACLevel[scan_pos] = level;
     ACRun  [scan_pos] = run;
     ++scan_pos;
      run=-1;
      nonzero=TRUE;
   }
}
```

```
...
CMPLT   .L2     B6,2,B0
[ B0]   B       .S1     L2
|| [!B0]   LDW     .D2T2   *B13,B4
|| [ B0]   SUB     .D1     A11,1,A0
|| [!B0]   ZERO    .L1     A0

  [ A0]   B       .S2     L1
  [ B0]   SHL     .S1     A7,4,A3
  [ B0]   ADD     .D1     A8,A3,A3

  [ B0]   ADDAH   .D1     A4,A3,A5
|| [!B0]   SHL     .S1     A7,4,A3
|| [ B0]   ZERO    .L1     A6

  [ B0]   STH     .D1T1   A6,*A5
|| [!B0]   ADD     .S1     A8,A3,A3
|| [!B0]   ADD     .D2     B12,B4,B4
|| [ B0]   SUB     .L1     A11,1,A11
...
```

Loop 내에 branch 명령어가 사용 => software pipeline 안됨

Cycles/ Instructions = 633 / 714

```
for(coeff_ctr=0 ; coeff_ctr<16; coeff_ctr++)
{
    ij = *zz ++;
    i = ij &0x3;     j = ij >>2;
    run++;     ilev=0;
    level = level_buf[i][j];
    if(level >1)      {
    *coeff_cost += MAX_VALUE;
     level = sign2(level,img->m7[i][j]);
    }
    if(level>1) {
     ACLevel[scan_pos] = level;
     ACRun  [scan_pos] = run;
     ++scan_pos;
      run=-1;
      nonzero=TRUE;
} }
```

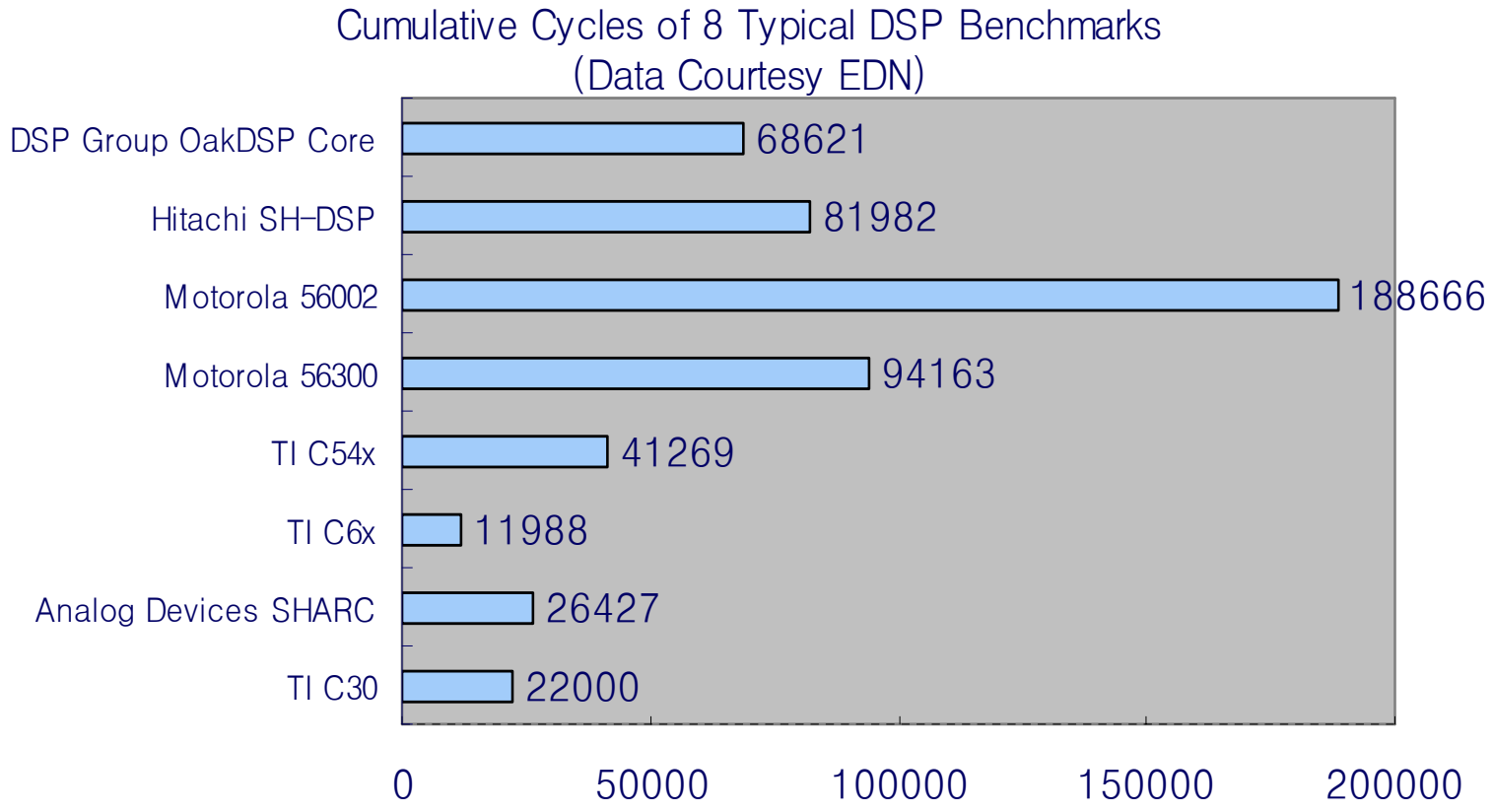
Branch 명령어의 사용이 조건문 내의 연산이 복잡해서 컴파일러가 잘 지원을 해주지 않는다. 조건문을 간소화 시켜, 컴파일러의 작업을 도와준다.

Cycles/Instructions = 607/625

C62xx Performance

Algorithm	‘C6x @ 200 MHz	Typical DSP @ 60 MHz	C6x/Typical Ratio	Clock Norm’d Ratio
256 FFT	14.0 us	199 us	14 : 1	4.2 : 1
8x8 DCT	1.14 us	15.3 us	13.4 : 1	4.02 : 1
Viterbi IS54 (89 terms)	29.5 us	315 us	10.7 : 1	3.21 : 1
24 tap LMS	0.21 us	1.9 us	9 : 1	2.7 : 1
8 biquads IIR	0.15 us	1.3 us	8.9 : 1	2.67 : 1
64 point 24 tap FIR	3.9 us	31 us	8 : 1	2.4 : 1

C6201 Compiler Efficiency (ver 1.0)
Cycle counts for unmodified C benchmark results



C6x family

- C62xx: Basic fixed-point VLIW
- C67xx: Floating-point VLIW
- C64xx: SIMD VLIW + Telecommunication acceleration unit (Turbo decoder).

Summary

- **'C6x VelociTI Advanced VLIW enables:**
  - Delivering 10x performance of any DSP on the market today
  - Shifting development paradigm from a hardware focus to a software focus
  - Reducing development time by half with new-generation tools designed for greatest ease of use and maximum optimization
  - Reducing system cost by half for multi-channel/multi-function applications
- **Reference:**

  http://www.ti.com/sc/docs/products/dsp/c6000/index.htm

- Schedule
- April 16: Multiprocessor DSP, due of HW #4
- April 18: OpenMP (by Youngjune)
- April 23: Midterm, Homework 5 will be given (C64 programming)
- April 25: DSP_VLSI

HW1

- **64 tap linear phase FIR filter**
  - Linear phase
  - data arrangement – store two 16 bit data into a 32bit memory.
  - Software pipelining, loop unrolling.
  - Intrinsic (saturation, rounding)
  - C code programming (use instrinsic), after compilation, compare with the theoretical bounds. Discuss the theoretical bounds from the resource limitation.
  - Show the number of cycles as a function of the filter order (16, 32, 64, 128)
  - Wonchul gives the Naïve C code. Coefficients are constant

Homework2

- **64 tap adaptive filter**
  - data arrangement
  - dependency problem for coefficients update
  - Software pipelining, loop unrolling.
  - Intrinsic (saturation, rounding)
  - C code programming (use instrinsic), compile and compare with the theoretical limit.
  - Show the number of cycles as a function of the filter order (16, 32, 64, 128)
  - Wonchul gives the Naïve C code.
    Coefficients are constant