计算优化-并行

李清安

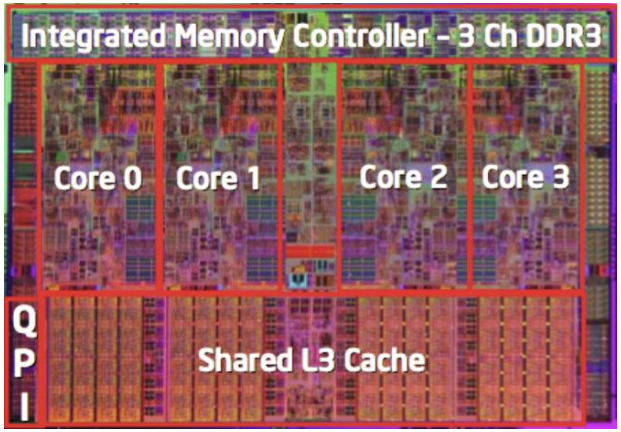
CPU并行优化

- 优化1：指令内并行
- 优化2：指令级并行
- 优化3：线程并行——GPU

指令内并行

- 8086，1978
  - First 16-bit Intel processor
  - 29 K transistors, 5-10 MHz
- 386, 1985
  - First 32-bit Intel processor, IA32
  - 275 K transistors, 16-33 MHz
- Pentium 4E, 2004
  - First 64-bit Intel X86 processor, X86-64
  - 125 M transistors, 2800-3800 MHz
- Core 2, 2006
  - First multi-core Intel processor
  - 291 M transistors, 1060-3500 MHz

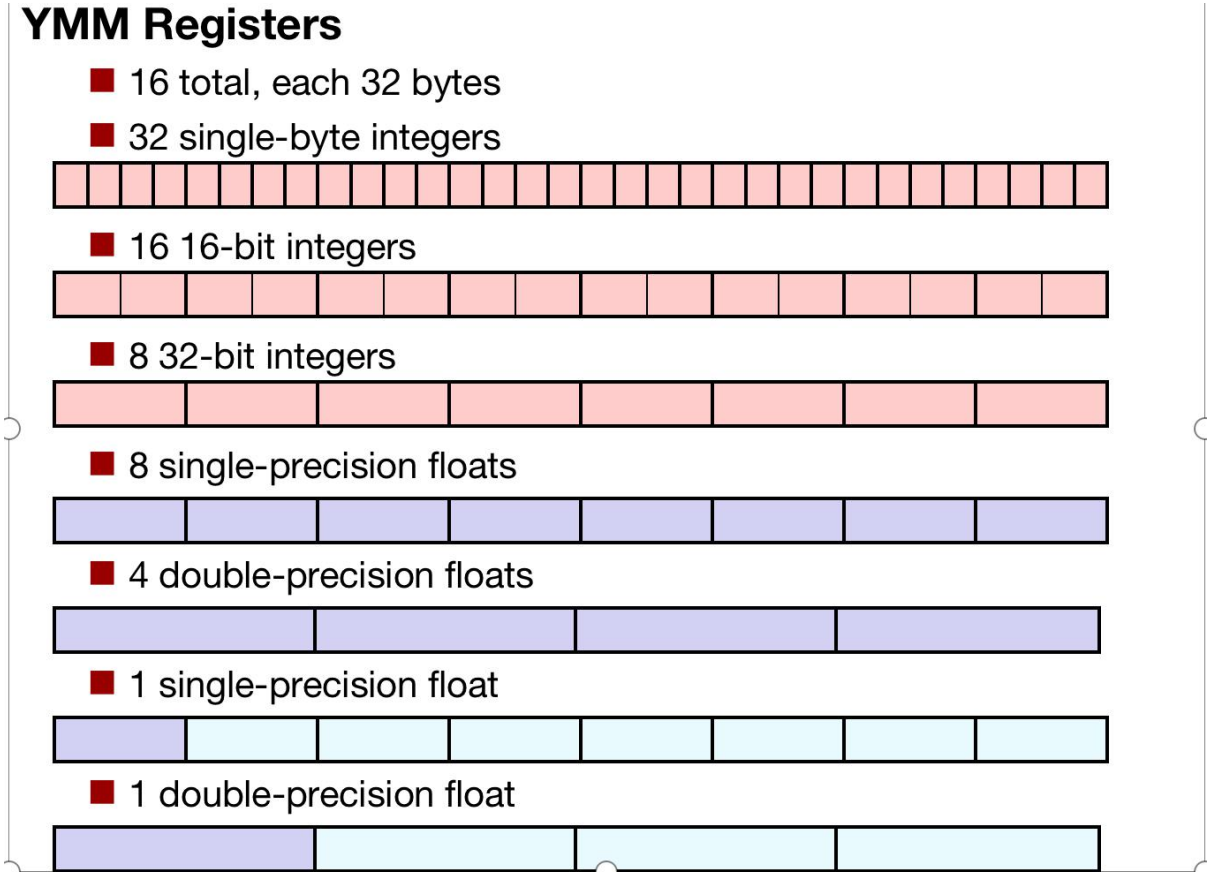
- Machine Evolution
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2001 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M



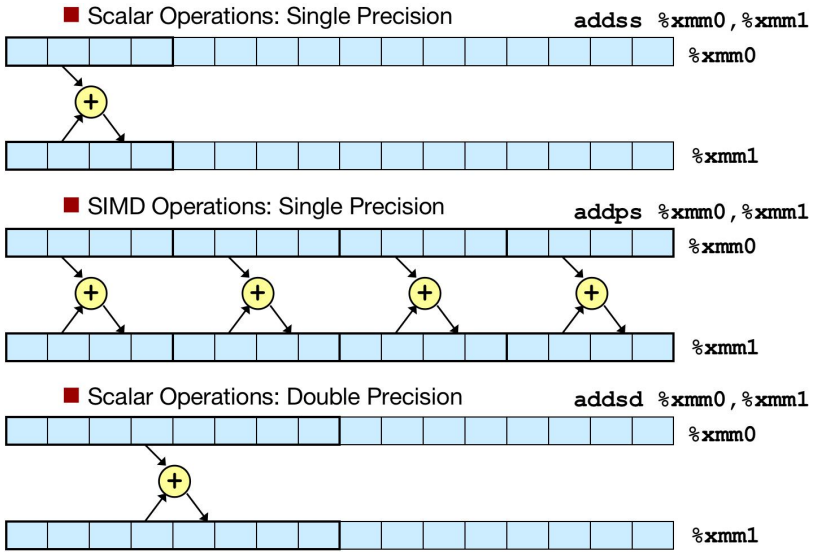
- Added Features
  - **Instructions to support multimedia operations**
  - Instructions to enable more efficient conditional operations
  - **Transition from 32 bits to 64 bits**
  - More cores

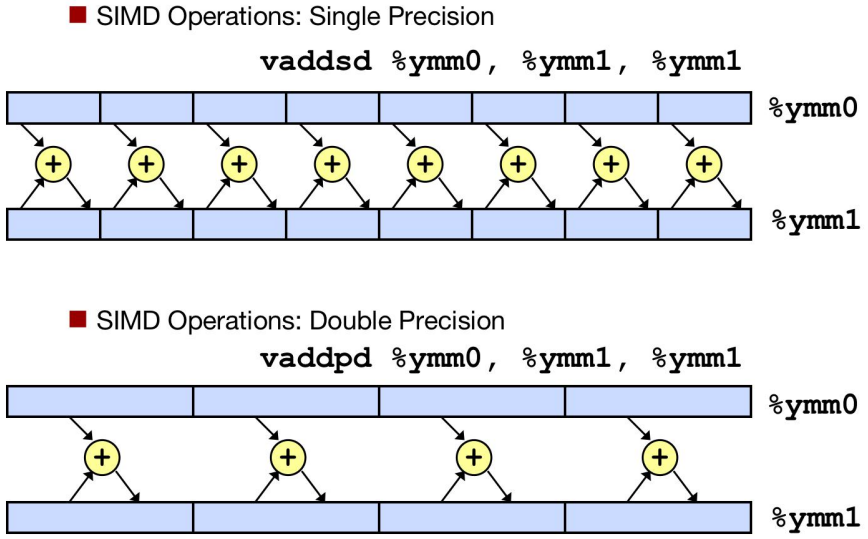
- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
- Example: 128-bit adder:
  - Sixteen 8-bit adds
  - Eight 16-bit adds
  - Four 32-bit adds
- Also called
  - data-level parallelism
  - vector parallelism
  - Single Instruction Multiple Data (SIMD)

- Intel XMM 256 bits * 16 -> YMM 512 * 16



- 向量指令


Scalar Operations: Single Precision
addss %xmm0,%xmm1
%xmm0
%xmm1
SIMD Operations: Single Precision
addps %xmm0,%xmm1
%xmm0
%xmm1
Scalar Operations: Double Precision
addsd %xmm0,%xmm1
%xmm0
%xmm1


SIMD Operations: Single Precision
vaddsd %ymm0, %ymm1, %ymm1
%ymm0
%ymm1
SIMD Operations: Double Precision
vaddpd %ymm0, %ymm1, %ymm1
%ymm0
%ymm1

- 向量指令的实现

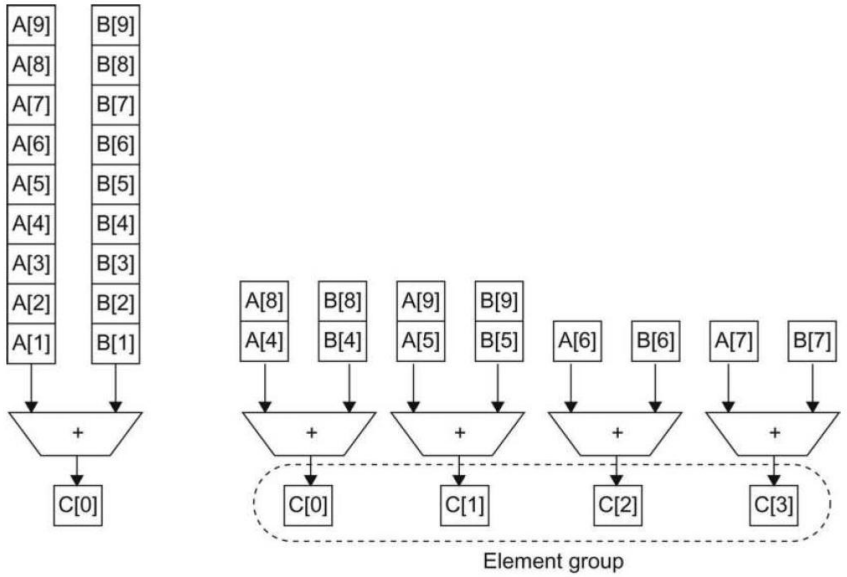


FIGURE 6.3 Using multiple functional units to improve the performance of a single vector add instruction, C =A +B.

The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines or lanes and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four lanes.

Example

- 该函数实现了什么功能？

```
1.  void dgemm (size_t n, double* A, double* B, double* C)
2.  {
3.    for (size_t i = 0; i < n; ++i)
4.      for (size_t j = 0; j < n; ++j)
5.      {
6.        double cij = C[i+j*n]; /* cij = C[i][j] */
7.        for(size_t k = 0; k < n; k++ )
8.          cij += A[i+k*n] * B[k+j*n]; /*cij+=A[i][k]*B[k][j]*/
9.        C[i+j*n] = cij; /* C[i][j] = cij */
10.     }
11. }
```

- 汇编：
  - 1. vmovsd (%r10),%xmm0 # Load 1 element of C into %xmm0
  - 2. mov %rsi,%rcx # register %rcx = %rsi
  - 3. xor %eax,%eax # register %eax = 0
  - 4. vmovsd (%rcx),%xmm1 # Load 1 element of B into %xmm1
  - 5. add %r9,%rcx # register %rcx = %rcx + %r9
  - 6. **vmulsd** (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
  - 7. add $0x1,%rax # register %rax = %rax + 1
  - 8. cmp %eax,%edi # compare %eax to %edi
  - 9. **vaddsd** %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
  - 10. jg 30 <dgemm+0x30> # jump if %eax > %edi
  - 11. add $0x1,%r11d # register %r11 = %r11 + 1
  - 12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element

- Note:
  - array C and array A should be column-major layout

```
1.  //include <x86intrin.h>
2.  void dgemm (size_t n, double* A, double* B, double* C)
3.  {
4.    for ( size_t i = 0; i < n; i+=4 )
5.      for ( size_t j = 0; j < n; j++ ) {
6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.        for( size_t k = 0; k < n; k++ )
8.          c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                  _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                 _mm256_broadcast_sd(B+k+j*n)));
11.       _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.     }
13. }
```

- 优化后汇编
  - 1. **vmovapd** (%r11),%ymm0 # Load 4 elements of C into %ymm0
  - 2. mov %rbx,%rcx # register %rcx = %rbx
  - 3. xor %eax,%eax # register %eax = 0
  - 4. **vbroadcastsd** (%rax,%r8,1),%ymm1 # Make 4 copies of B element
  - 5. add $0x8,%rax # register %rax = %rax + 8
  - 6. **vmulpd** (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1,4 A elements
  - 7. add %r9,%rcx # register %rcx = %rcx + %r9
  - 8. cmp %r10,%rax # compare %r10 to %rax
  - 9. **vaddpd** %ymm1,%ymm0,%ymm0 # Parallel add %ymm1, %ymm0
  - 10. jne 50 <dgemm+0x50> # jump if not %r10 != %rax
  - 11. add $0x1,%esi # register % esi = % esi + 1
  - 12. vmovapd %ymm0,(%r11) # Store %ymm0 into 4 C elements
- **这种优化谁来做?**
- http://const.me/articles/simd/simd.pdf

- 这个函数计算了什么？

```
template<>
float dotProduct<eDotProductAlgorithm::SseDpPs>( const float* p1, const float* p2, size_t count )
{
        assert( 0 == count % 4 );

        __m128 acc = _mm_setzero_ps();
        const float* const p1End = p1 + count;
        for( ; p1 < p1End; p1 += 4, p2 += 4 )
        {
                // Load 2 vectors, 4 floats / each
                const __m128 a = _mm_loadu_ps( p1 );
                const __m128 b = _mm_loadu_ps( p2 );
                // Compute dot product of them. The 0xFF constant means "use all 4 source lanes, an
                const __m128 dp = _mm_dp_ps( a, b, 0xFF );
                acc = _mm_add_ps( acc, dp );
        }
        // By the way, the intrinsic below compiles into no instructions.
        // When a function is returning a float, modern compilers pass the return value in the lowe
        return _mm_cvtss_f32( acc );
}
```

- 这个函数?

```
// Convert 4 pixels into grayscale, return 4-wide int32 vector.
template<bool fma>
inline __m128i grayscale_float4( const __m128i *source )
{
        __m128i pixels = _mm_loadu_si128( source );
        const __m128 red = makeFloats( pixels, 0xFF );
        const __m128 green = makeFloats( pixels, 0xFF00 );
        const __m128 blue = makeFloats( pixels, 0xFF0000 );
        __m128 res = _mm_mul_ps( red, _mm_set1_ps( mulRedFloat ) );
        if constexpr( fma )
        {
                // Using FMA to multiply + accumulate
                res = _mm_fmadd_ps( green, _mm_set1_ps( mulGreenFloat / 0x100 ), res );
                res = _mm_fmadd_ps( blue, _mm_set1_ps( mulBlueFloat / 0x10000 ), res );
        }
        else
        {
                // No FMA, doing the same math as above with separate add and mul instructions.
                res = _mm_add_ps( res, _mm_mul_ps( green, _mm_set1_ps( mulGreenFloat / 0x100 ) ) );
                res = _mm_add_ps( res, _mm_mul_ps( blue, _mm_set1_ps( mulBlueFloat / 0x10000 ) ) );
        }
        return _mm_cvtps_epi32( res );
}
```

- 你能想到其他什么场景下需要使用这些指令?

- 随着应用发展，还需要哪些指令?

- 向量指令?
- 矩阵指令?
- 张量指令?
- GPU? TPU? NPU?
  - 人工智能应用如何优化?

- GPU block diagram

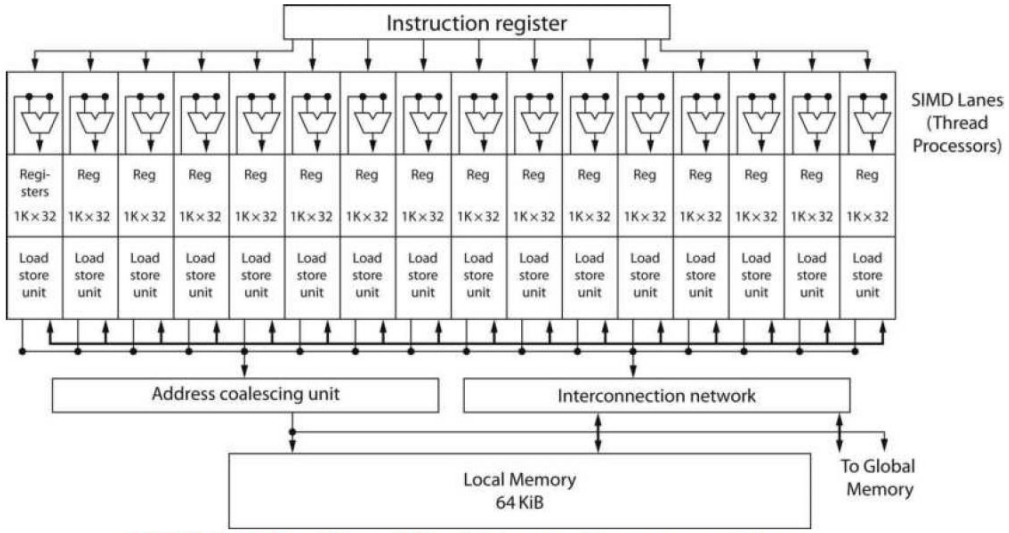


FIGURE 6.9 Simplified block diagram of the datapath of a multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has many independent SIMD threads that it chooses from to run on this processor.

- GPU

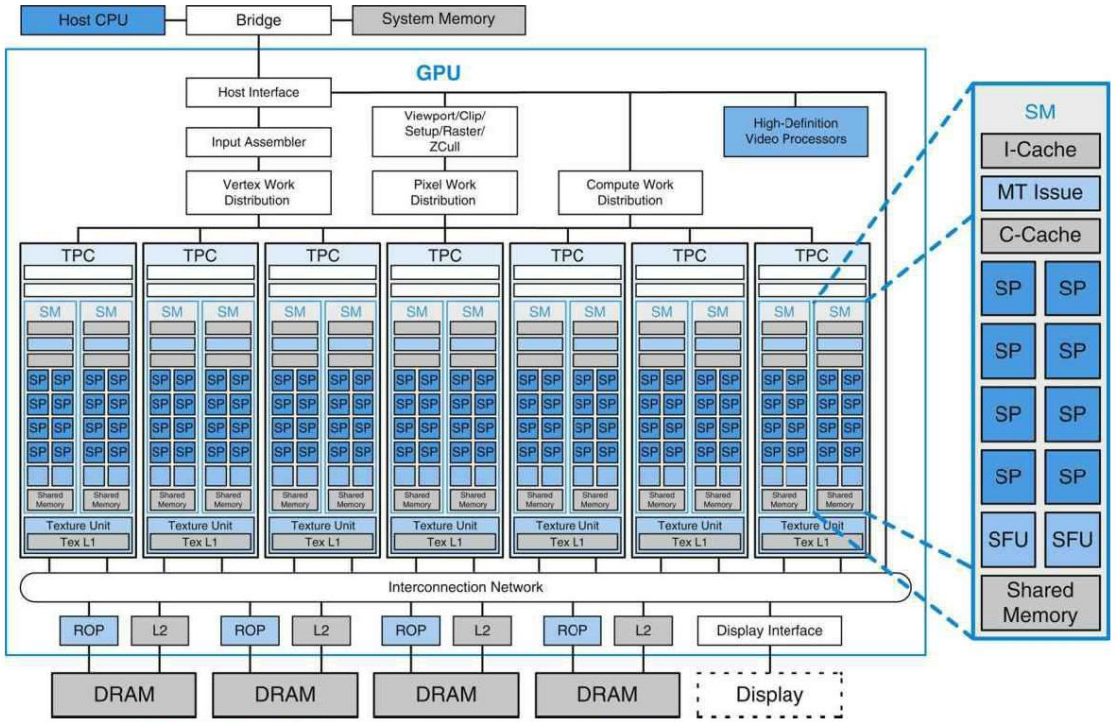
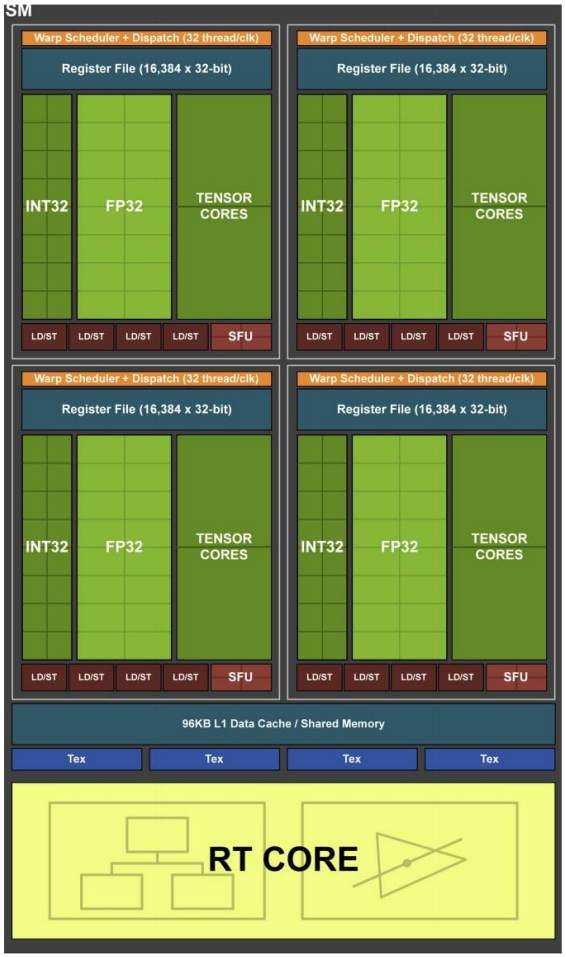


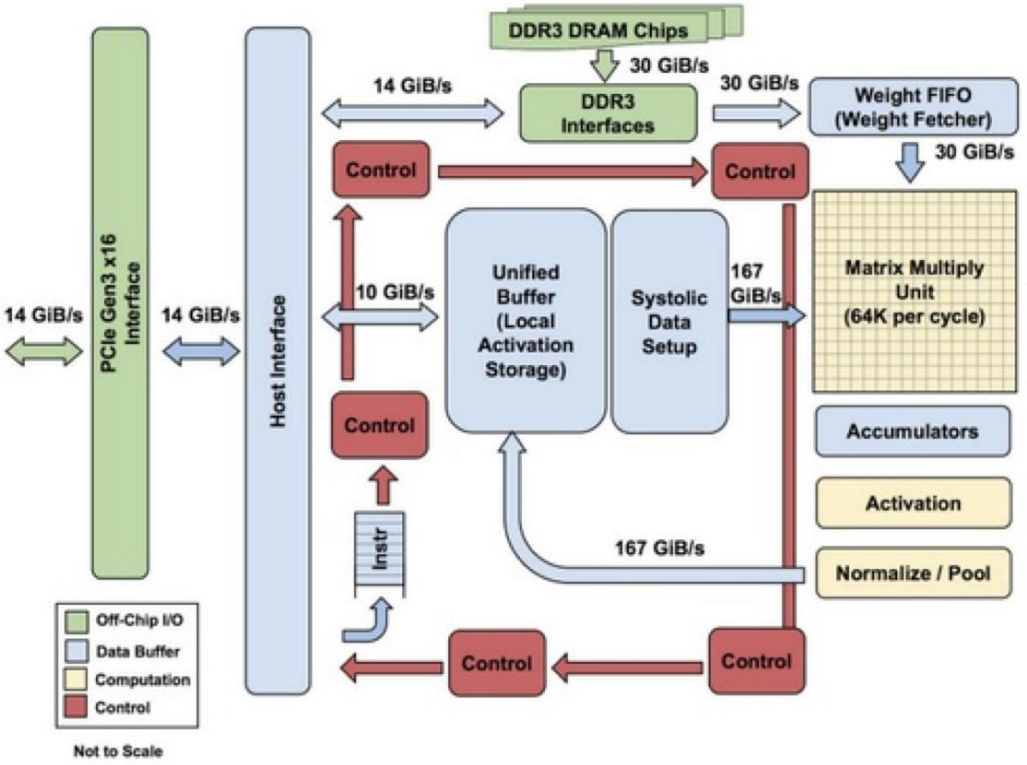
FIGURE B.2.5 Basic unified GPU architecture. Example GPU with 112 *streaming processor* (SP) cores organized in 14 *streaming multiprocessors* (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two *special function units* (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

- 64个Cuda核心FP32
- 64个INT32，同步更新地址
- 8个张量核
  - 单核单cycle：64个FP16**乘-加运算**
- 256K 寄存器，96K L1缓存



TPU

- 矩阵乘法单元
- 激活
- 归一化/池化

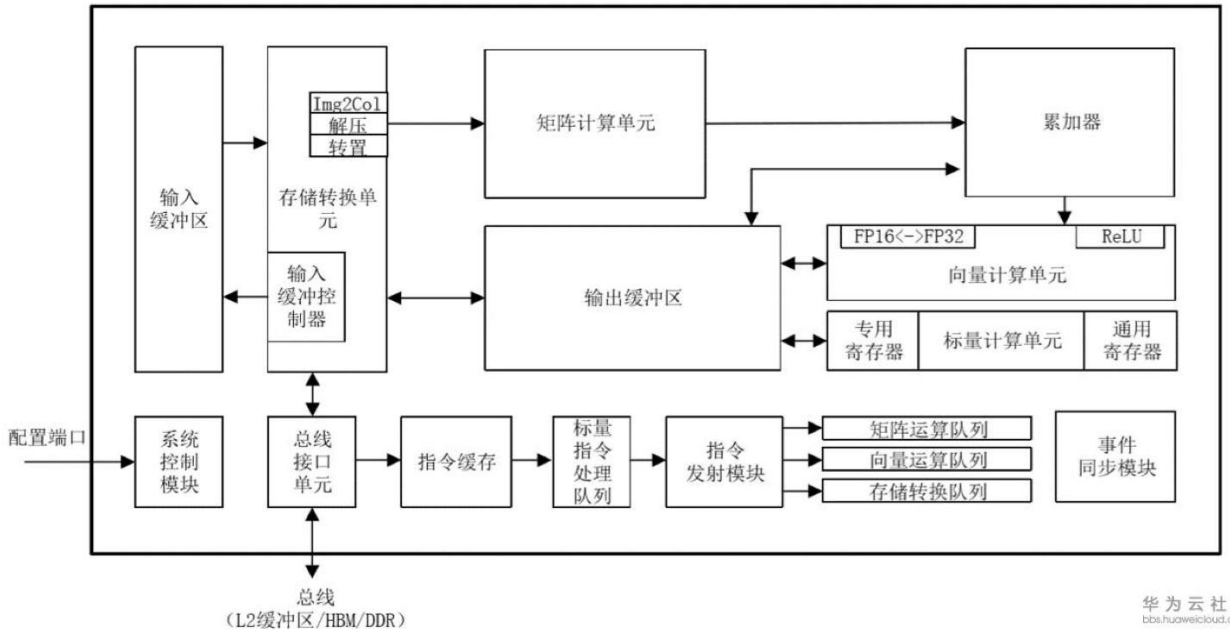


寒武纪芯片

指令类型		例子	操作对象
控制指令		跳转（JUMP），条件分支（CB）	寄存器（标量值），立即数
数据传输指令	矩阵（Matrix）	矩阵取（MLOAD）/存（MSTORE）/移动（MMOVE）	寄存器（矩阵地址/大小，标量值）立即数
	向量（Vector）	向量取（VLOAD）/存（VSTORE）/移动（VMOVE）	寄存器（向量地址/大小，标量值）立即数
	标量（Scalar）	标量取（SLOAD）/存（SSTORE）/移动（SMOVE）	寄存器（标量值），立即数
计算指令	矩阵（Matrix）	矩阵乘向量（MMV），向量乘矩阵（VMM），矩阵乘标量（MMS），外积（OP），矩阵相加（MAM），矩阵相减（MSM）	寄存器（矩阵或向量地址/大小，标量值）
	向量（Vector）	向量基本运算［加（VAV）、减（VSV）、乘（VMV）、除（VDV）］，向量超越函数［指数（VEXP）、对数（VLOG）］，内积（IP），随机向量（RV），向量最大值（VMAX），向量最小值（VMIN）	寄存器（向量地址/大小，标量值）
	标量（Scalar）	标量基本运算，标量超越函数	寄存器（向量地址/大小，标量值）
逻辑指令	向量（Vector）	向量比较［大于（VGT）、等于（VE）］，向量逻辑操作［与（VAND）、或（VOR）、取反（VNOT）］，向量最值归约（VGTM）	寄存器（向量地址/大小，标量值）
	标量（Scalar）	标量比较，标量逻辑运算	寄存器（标量值），立即数

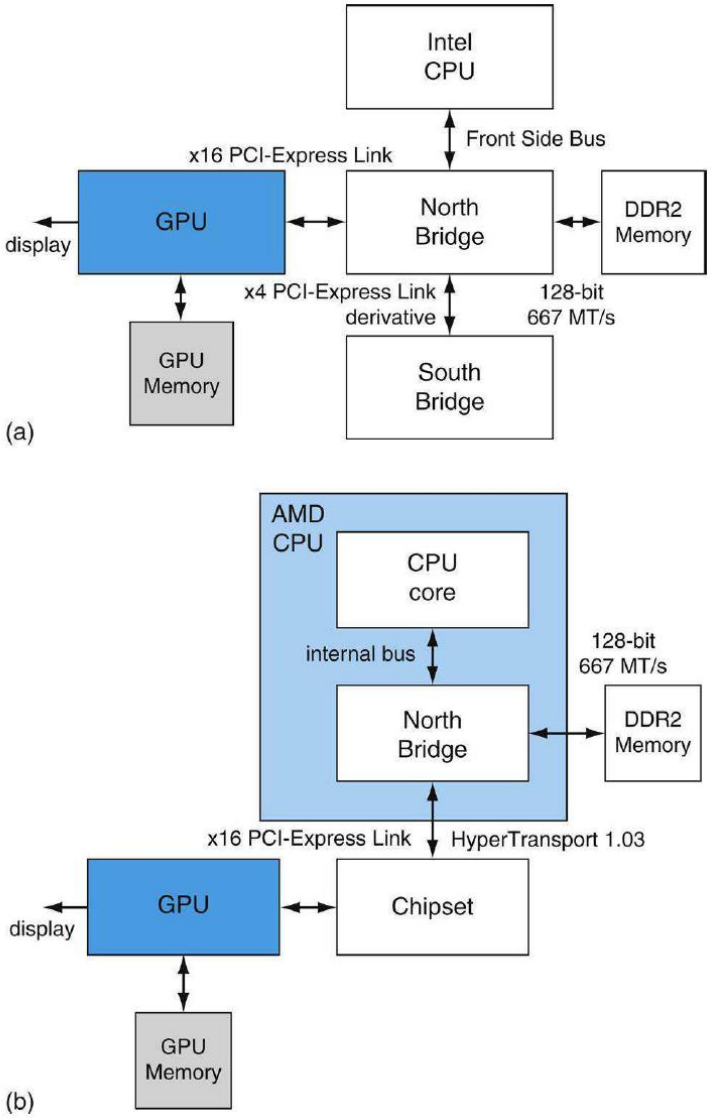
华为NPU-昇腾

- 矩阵计算单元
- 向量计算单元
- 标量计算单元



- 要求：指令内并行要求数据的规则访问
  - 8-bit -> 64-bit
  - SIMD: 数组、矩阵
- 这个要求是否是太高？

- CPU + GPU异构计算

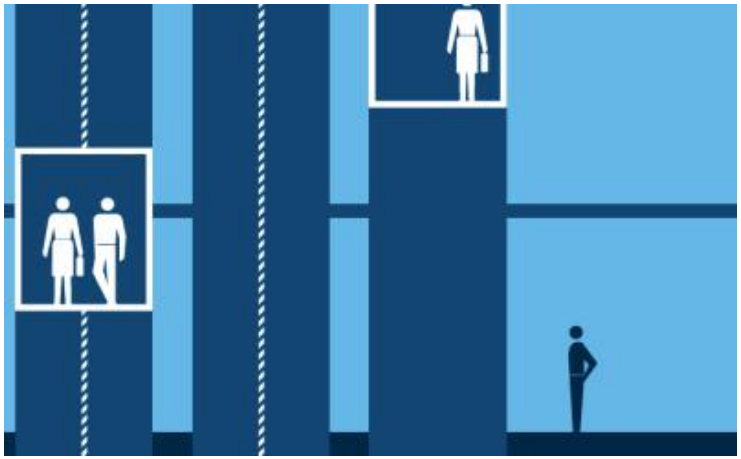

Intel
CPU
Front Side Bus
x16 PCI-Express Link
GPU
North
Bridge
DDR2
Memory
display
x4 PCI-Express Link
derivative
128-bit
667 MT/s
GPU
Memory
South
Bridge
(a)
AMD
CPU
CPU
core
internal bus
128-bit
667 MT/s
North
Bridge
DDR2
Memory
x16 PCI-Express Link
HyperTransport 1.03
GPU
Chipset
display
GPU
Memory
(b)

CPU并行优化

- 优化1：指令内并行
  - 指令内操作多个数据
- 优化2：指令级并行
  - 同时执行多条指令
- 优化3：线程并行——GPU

流水线并行

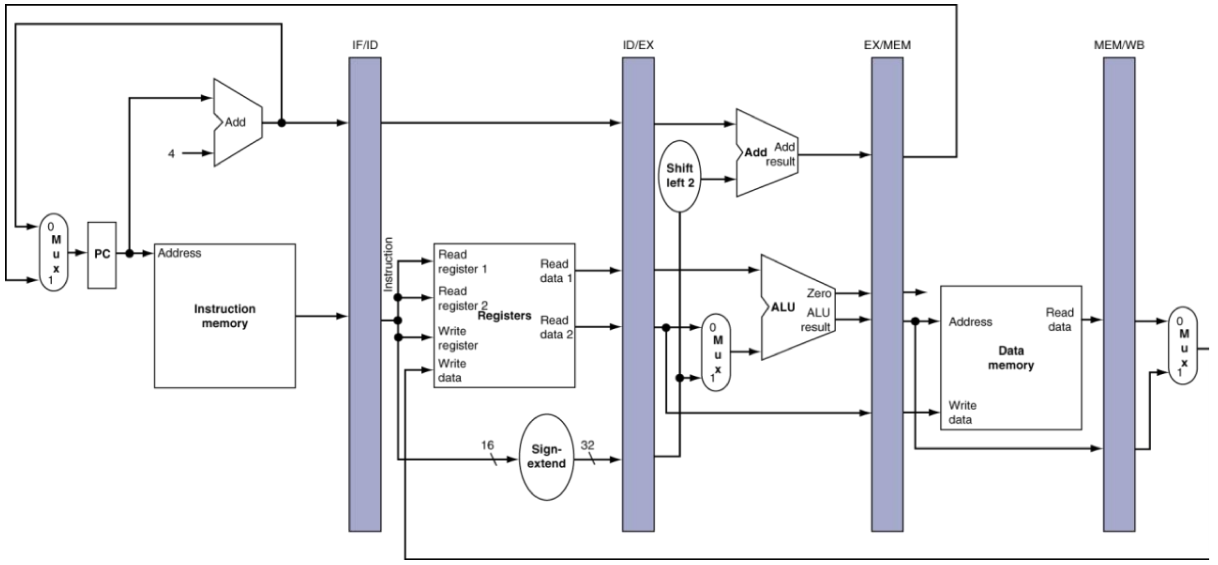
- 垂直电梯 VS 扶手电梯 （不考虑启动时间）
  - 4秒一层楼，12人
  - 吞吐量：12p/4s = 3 p/s
- 扶手电梯
  - 30级台阶，每台阶2人，12s
  - 吞吐量：30*2p/12s= 5p/s （满流水）
- 真实差距?

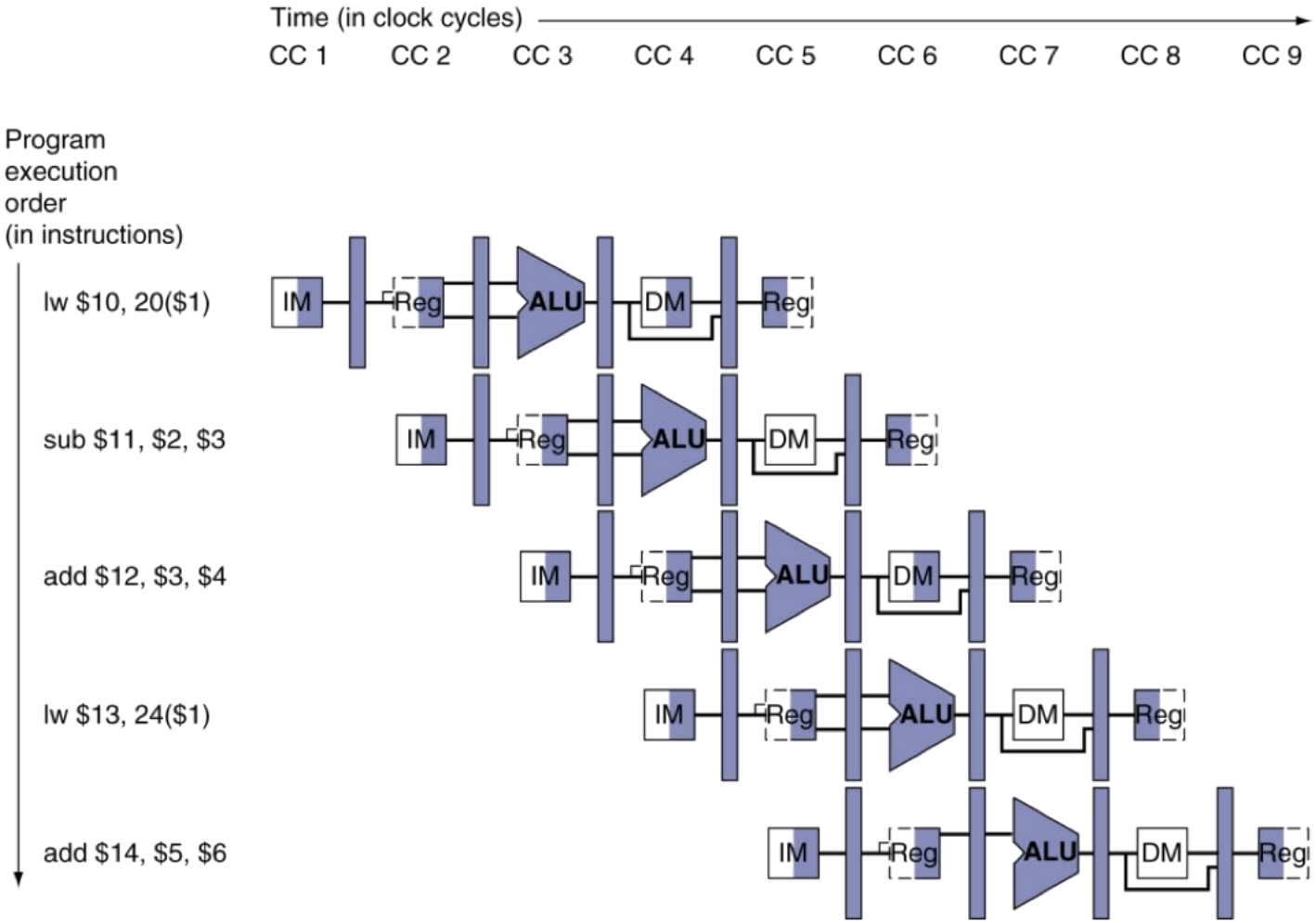


流水线并行

- 非流水线CPU：
  - 每条指令独占整个数据通道
  - 每个时钟完成一条指令，但是时钟周期很长
- 流水线CPU
  - 加速原理：
    - 将一个过程拆分成多个阶段
    - 最长阶段决定时钟周期，即时钟频率
- 理想加速比？
  - 4s一层楼？每级站12人
  - 30级台阶 -> 30*12p/4s= 90 p/s ： 30X

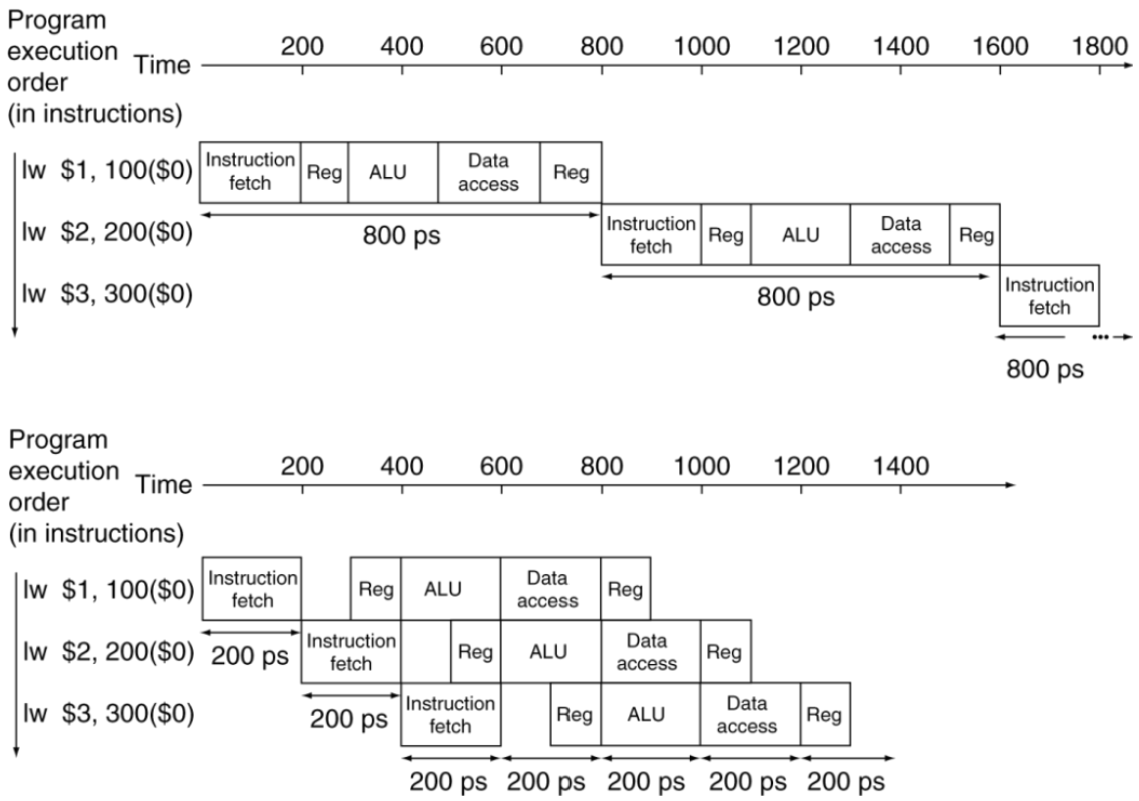
- 五阶段流水
  - IF: Instruction fetch from memory
  - ID: Instruction decode & register read
  - EX: Execute operation or calculate address
  - MEM: Access memory operand
  - WB: Write result back to register




Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
IM
Reg
ALU
DM
Reg

障碍1：各流水阶段不平衡

- 800 ps vs 200 ps
  - Reg：100 ps
  - ALU：200ps



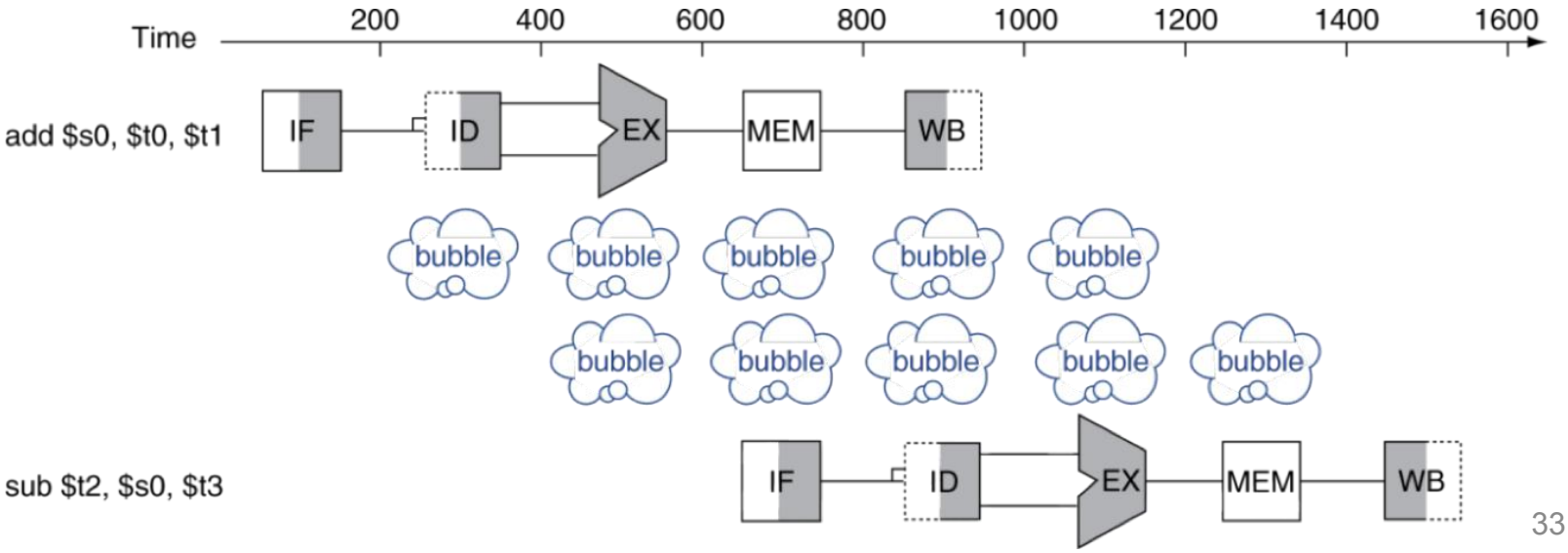
- 为什么要内存对齐?

障碍2：硬件资源竞争

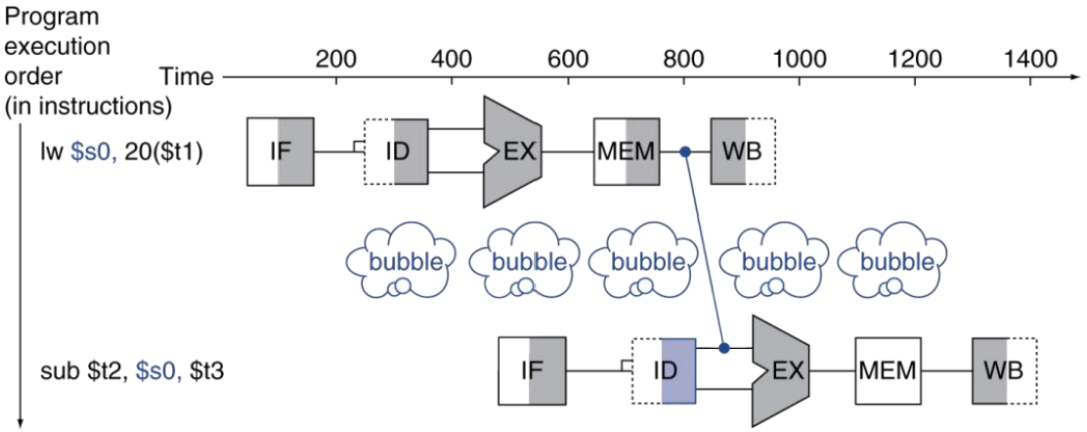
- 结构hazards
  - 硬件逻辑竞争：比如，两条指令访问同一个寄存器；load和store竞争memory bus
- 如何缓解？
  - 加硬件资源
    - 单一内存 -> 哈佛体系结构
    - Intel: L1 cache
      - Inst cache
      - Data cache
    - > sysctl -a |grep “cache”
  - 超标量

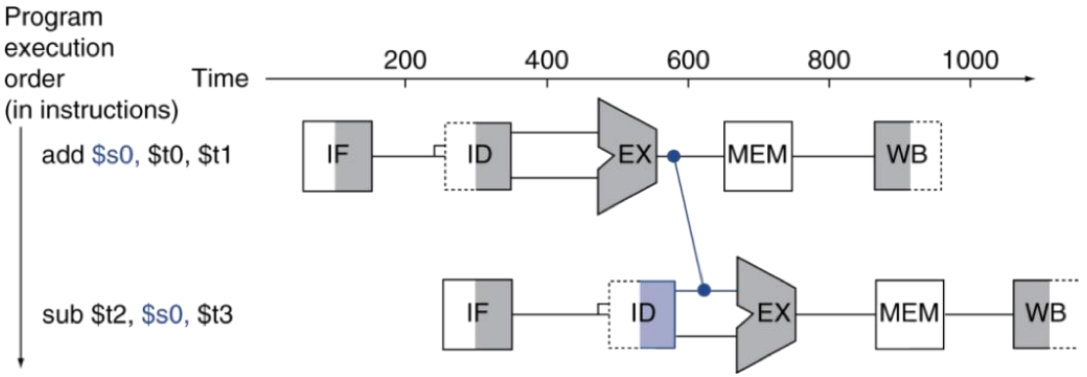
障碍3: 依赖

- 数据hazard
  - 下一条指令的输入，依赖于上一条指令的输出
  - 需要等待前面指令完成，才能开始下一条指令
  - 例子:
    - add $s0, $t0, $t1
    - sub $t2, $s0, $t3

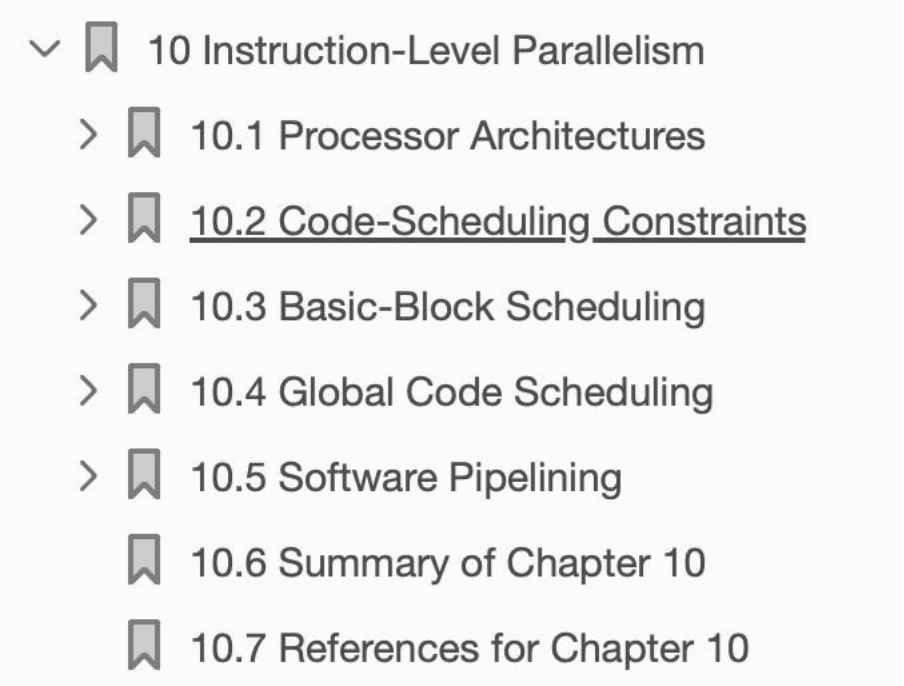


- 如何缓解数据依赖？
  - 硬件
    - Bypassing
    - Stall



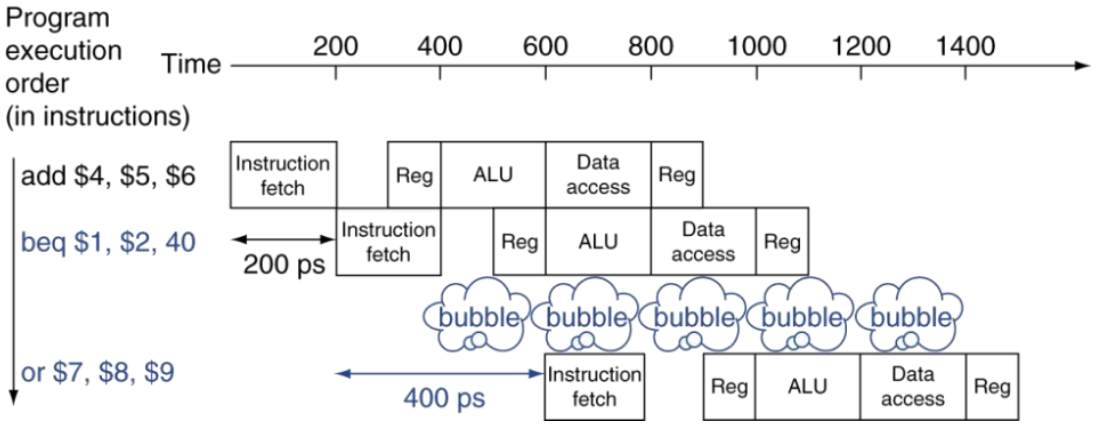


- 如何缓解数据依赖？
  - 软件
    - Instruction scheduling
    - 循环展开
    - 谁来做？
  - 参考
    - Compiler: Principles, Techniques, and Tools，龙书第10章



障碍3: 依赖

- 控制hazard
  - 指令跳转行为需要等待前面指令完成
  - 例子
    - 指令beq完成EX阶段后，才能确定下一条指令的地址
  - 分支对流水的影响有多大?
    - Intel i5，31级流水
    - 平均5条指令就有一条分支
    - 据此估算分支的影响?



- 如何缓解控制依赖
  - 指令调度（软件实现）
    - **循环展开的优势？**
  - 分支预测（硬件实现）
    - 选出概率最高的分支，直接进入流水
    - 静态预测（loop feature）、动态预测（history）
    - 如果预测失败？

```
const char *home_dir ;

home_dir = getenv("HOME");
if (likely(home_dir))
    printf("home directory: %s\n", home_dir);
else
    perror("getenv");
```

  - 能否软件实现分支预测？
    - 静态、动态、程序员辅助？如果预测失败？

指令调度与软件流水

```
for i = 1 to N
    A(i);    // load data i
    B(i);    // compute data i
    C(i);    // store data i
end
```

- 指令调度
- 指令调度与循环
  - 循环展开——software pipelining
    - 减少分支执行
    - 改善指令调度的优化空间
  - 例子：
    - 假设A、B、C指令latency：3 cycle
    - A -> B -> C
    - 优化效果?
  - 问题：
    - 增加硬件hazards？寄存器压力？
    - 体系结构支持?

```
A(1); B(1); C(1); A(2); B(2); C(2); A(3); B(3); C(3) ...
```

```
for i = 1 to N step 3
    A(i);    // load data i
    A(i+1);    // load data i+1
    A(i+2);    // load data i+2
    B(i);    // compute data i
    B(i+1);    // compute data i+1
    B(i+2);    // compute data i+2
    C(i);    // store data i
    C(i+1);    // store data i+1
    C(i+2);    // store data i+2
end
```

- 参考：
  - https://en.wikipedia.org/wiki/Software_pipelining

流水线的讨论

- 限制：
  - 负载均衡？各流水阶段是否均衡
  - 依赖：流水停顿
  - 硬件资源竞争
    - 只有一个IF单元，每个cycle最多发射一条指令

流水线的讨论

- 流水线（pipeline）：
  - 更深的流水线 -> 更大的吞吐量?
  - 每个时钟发射多少条指令?
  - 伟大的工业制造发明
  - 你怎么看研究生流水线？阅读文献-idea-实验-写作-投稿
- 多发射（multiple issue）
  - 更多的硬件逻辑
  - 每个时钟发射多条指令
  - 理想情况下：4GHz 4-way multiple-issue, 16 BIPS, peak IPC = 4
  - 但是，有依赖的指令不能并行

多发射

- 静态多发射
  - 编译器负责识别指令之间的依赖，并将可并行的指令打包发射，Very Long Instruction Word (VLIW)
- 动态多发射
  - CPU负责识别指令之间的依赖，超标量（superscalar CPU）、乱序执行
  - 编译器能够处理更大的窗口，所以总能帮助
    - 软硬件结合优化
- 如果依赖太多？
  - 投机执行（speculation）：预测执行 + 回滚
    - 分支指令
    - load指令 after store
  - **投机执行 vs 分支预测？**

- 例子
  - 静态2发射通路
  - 64bit IF
  - Reg：4R+2W
    - ➢write-dependence?
  - 2 ALU

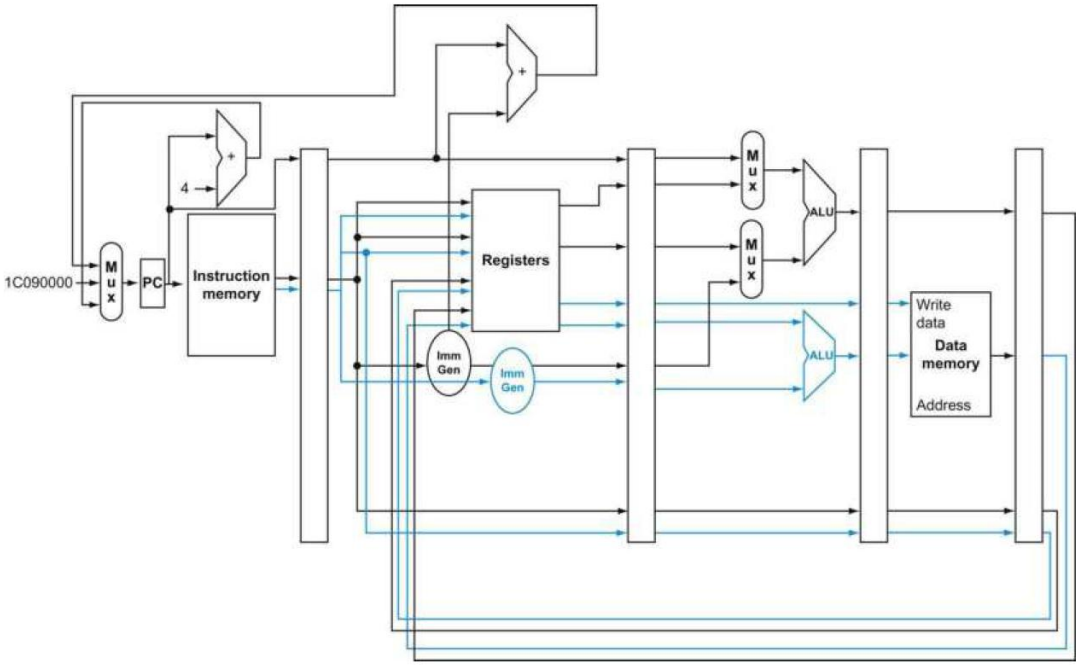


FIGURE 4.66 A static two-issue datapath.
The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

- 例子
  - 假设可同时发射ALU+Load
  - 编译器将ALU+Load调度一起
  - IF一次取指64bit（2条指令）
- 如果相邻指令不满足条件
  - 非ALU+Load
  - 有依赖
- 编译优化（指令调度）
  - 循环：循环展开
  - 如果潜力不够，填充nop

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 4.65 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

- 前面3条依赖t0
- 后面2条依赖s1

```
Loop: lw    $t0, 0($s1)    # $t0=array element
      addu  $t0,$t0,$s2# add scalar in $s2
      sw    $t0, 0($s1)# store result
      addi  $s1,$s1,-4# decrement pointer
      bne   $s1,$zero,Loop# branch $s1!=0
```

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

FIGURE 4.70 The scheduled code as it would look on a two-issue MIPS pipeline. The empty slots are no-ops.

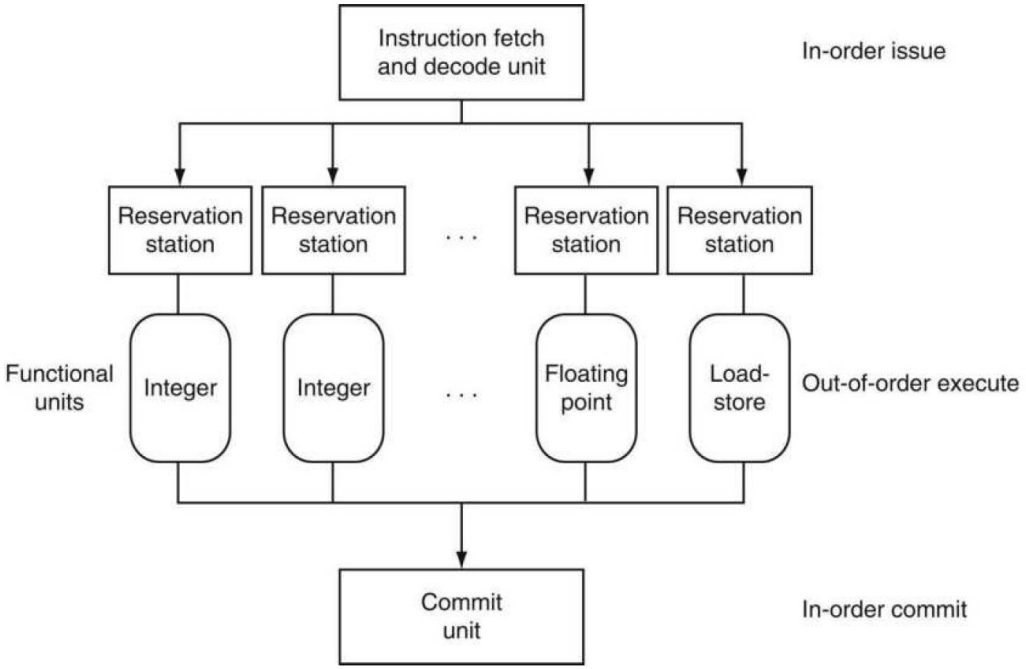
循环展开

	ALU or branch instruction		Data transfer instruction		Clock cycle
Loop:	addi	$s1,$s1,-16	lw	$t0, 0($s1)	1
			lw	$t1,12($s1)	2
	addu	$t0,$t0,$s2	lw	$t2, 8($s1)	3
	addu	$t1,$t1,$s2	lw	$t3, 4($s1)	4
	addu	$t2,$t2,$s2	sw	$t0, 16($s1)	5
	addu	$t3,$t3,$s2	sw	$t1,12($s1)	6
			sw	$t2, 8($s1)	7
	bne	$s1,$zero,Loop	sw	$t3, 4($s1)	8

FIGURE 4.71 The unrolled and scheduled code of Figure 4.70 as it would look on a static two-issue MIPS pipeline. The empty slots are no-ops. Since the first instruction in the loop decrements $s1 by 16, the addresses loaded are the original value of $s1, then that address minus 4, minus 8, and minus 12.

超标量

- 硬件实现多发射
  - 硬件调度
- Compiler can help
- Programmer can he



Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	8	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level cache/core	128–2048 KiB (shared)	256 KiB (per core)
3rd level cache (shared)	(platform dependent)	2–8 MiB

FIGURE 4.71 Specification of the ARM Cortex-A53 and the Intel Core i7 920.

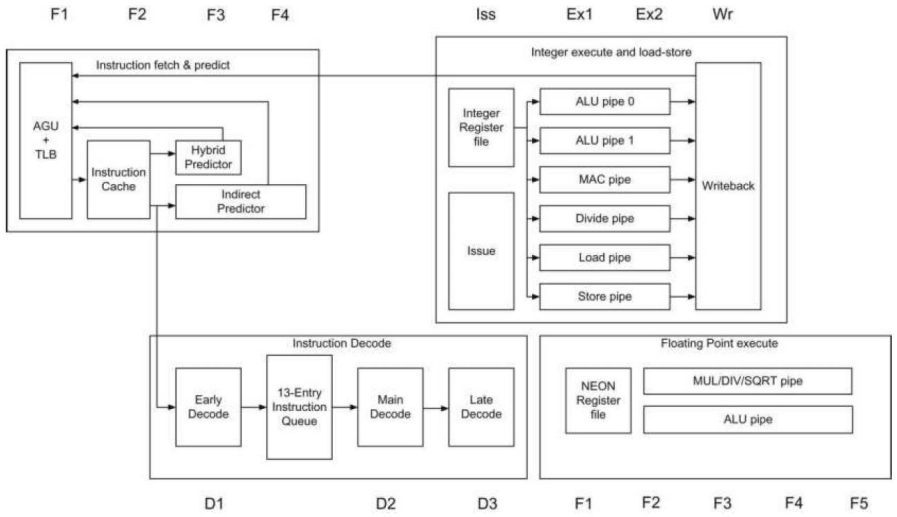


FIGURE 4.72 The Cortex-A53 pipeline.
The first three stages fetch instructions into a 13-entry instruction queue. The *Address Generation Unit* (AGU) uses a *Hybrid Predictor, Indirect Predictor*, and a *Return Stack* to predict branches to try to keep the instruction queue full. Instruction decode is three stages and instruction execution is three stages. With two additional stages for floating point and SIMD

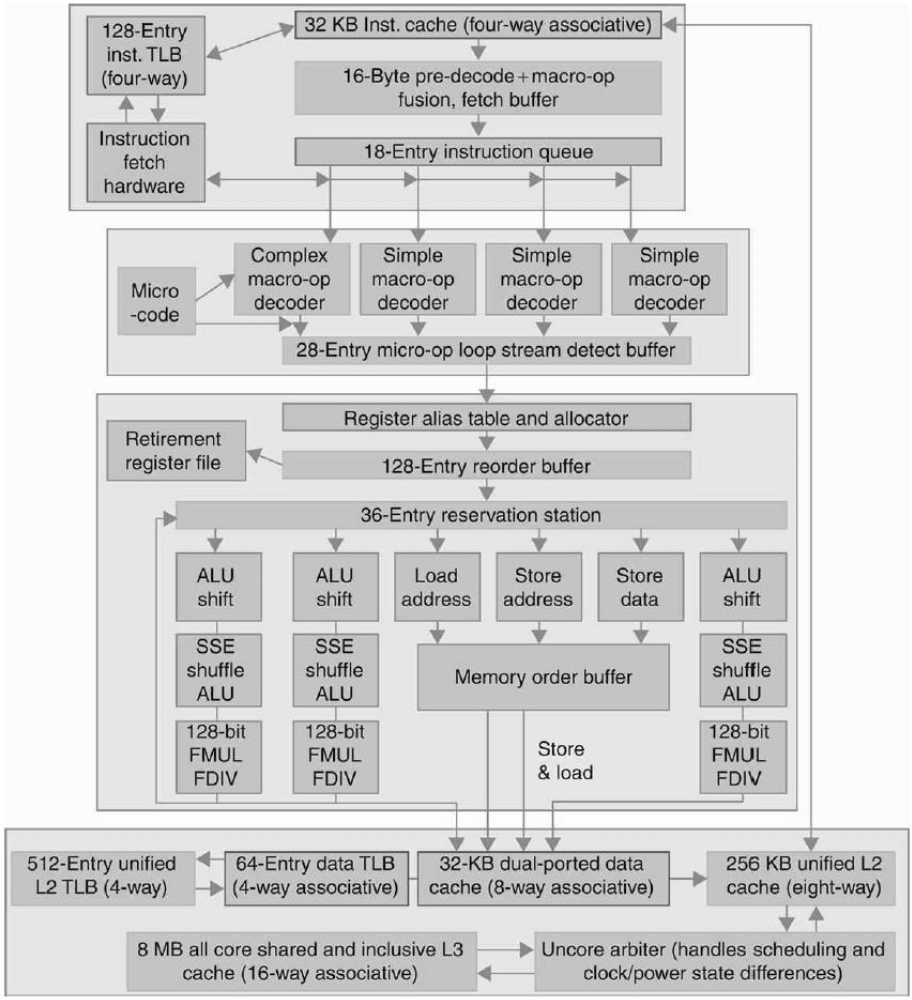


FIGURE 4.74 The Core i7 pipeline with memory components.
The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready micro-operation each clock cycle.

- How compiler helps?
  - 编译器能够处理更大的窗口，所以总能帮助
- How programmer helps?
  - See lec2.2.csapp-optimization slides from pp. 21

pipeline vs superscalar

- pipeline
  - only on data path
- superscalar
  - more datapath
  - but don't duplicate all, like regs, PCs

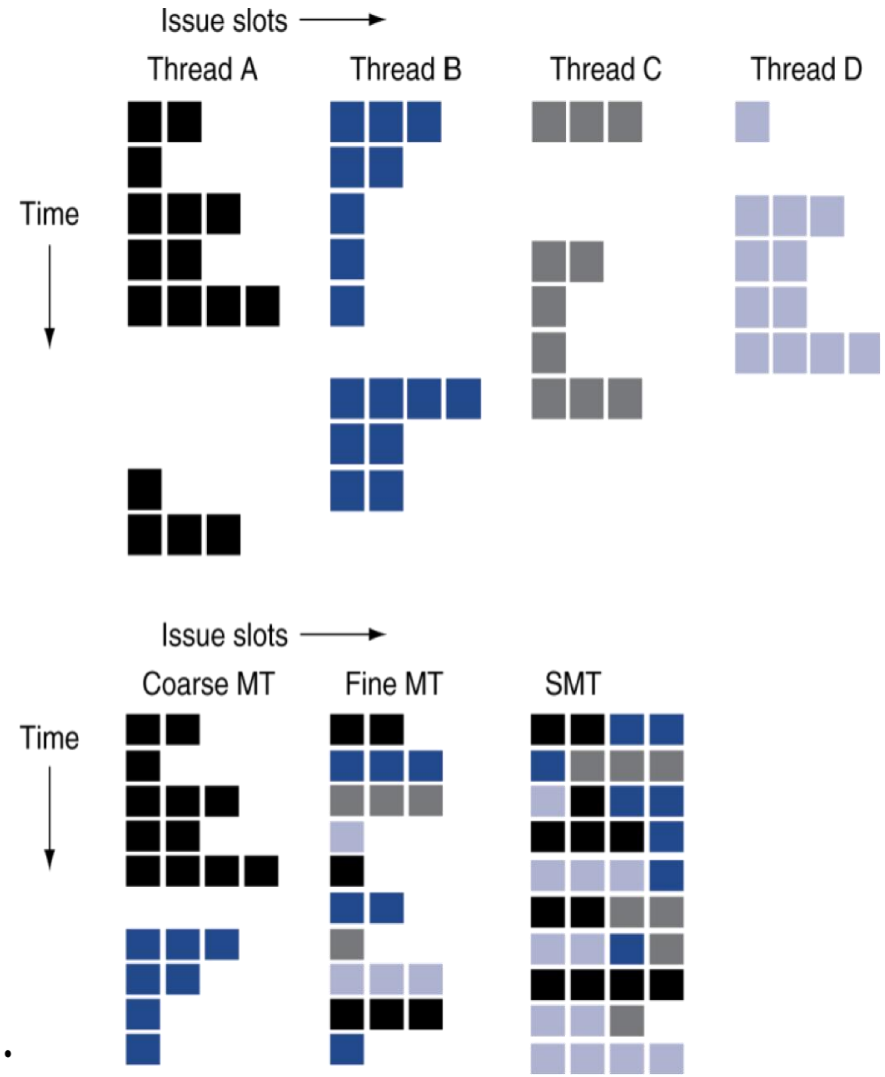
ILP小结

- **数组的规则访问，适合ILP**
- 由于数据依赖以及控制依赖的存在，ILP的作用有限
  - 硬件方法：pipelining和多发射
  - 软件方法：指令调度（+循环展开）
  - 软硬件方法：分支预测
- it works, but not as much as we'd like
  - 依赖的识别很困难
    - 指针别名
  - 应用的特点，限制了并行指令的窗口
    - 导致流水和多发射不能full
    - 尽管CPU支持4-6发射，很少应用能够实现IPC>2
  - 存储层次中的memory miss，导致流水大停顿；很难利用ILP弥补

more parallism: multi-threading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
  - Example: Intel Pentium-4 HT
- Two threads: duplicated registers, shared function units and caches


Issue slots
Thread A
Thread B
Thread C
Thread D
Time
Issue slots
Coarse MT
Fine MT
SMT
Time

superscalar vs multithreading

- superscalar
  - more datapath, but share PCs, regs, caches
- multithreading
  - more datapath, duplicate PCs, regs, but not caches
- What is next? More parallism?
  - multithreading in GPUs
  - multicore for general purpose computing
  - domain specific processor
    - DSP
    - NPU/TPU

Multi-cores for genereal purpose computing

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

FIGURE 4.70 Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power.

beyond计算优化

- cache?
- 内存
  - Instruction/Data Memory
- Memory是不是很慢?
  - latency：200-cycle