Computer Architecture

计 算 机 体 系 结 构

第7讲：ISA & ILP（5）

张献伟

xianweiz.github.io

DCS3013, 10/26/2022

Review: Loop Unrolling & Branch

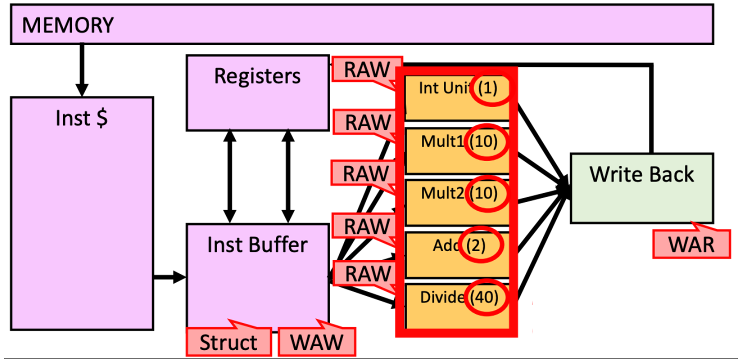
- Loop unrolling[循环展开]
  - Re-order instructions to transform
  - Loop unrolling to expose scheduling opportunities
  - Gains are limited by several factors

- Branch prediction[分支预测]
  - Predict how branches will behave to reduce stalls
  - Basic static predictor
  - Correlating predictors (a.k.a., two-level predictors)
    - *(m, n)*: last *m* branches, *n*-bit predictor for a single branch
  - Tournament predictors
    - Adaptively combining local and global predictors

Review: Dynamic Scheduling

- **Static** scheduling: in-order instruction issue and execution
  - If an inst is stalled in pipeline, no later insts can proceed
  - Loop unrolling: reduce stalls by separating dependent insts
    - Static pipeline scheduling by compiler
- **Dynamic** scheduling: in-order issue, OoO execution
  - Reorders the instruction execution to reduce the stalls while maintaining data dependence
  - OoO execution may introduce WAR and WAW hazards
    - Both can be avoided by register renaming
  - *ID* stage is split into two
    - Issue: decode insts, check for structural hazards
    - Read operands: wait until no data hazards, then read operands
  - Scoreboard: a technique for allowing insts to execute OoO when there are sufficient resources and no data dependences

Summary of Scoreboard



- Basic idea
  - Use scoreboard to track data dep. through register
- Main points of design
  - Instructions are sent to FU unit if there is no outstanding name dependence
  - RAW data dependence is tracked and enforced by scoreboard

How? Just stall the insts until the RAW hazard can be addressed.

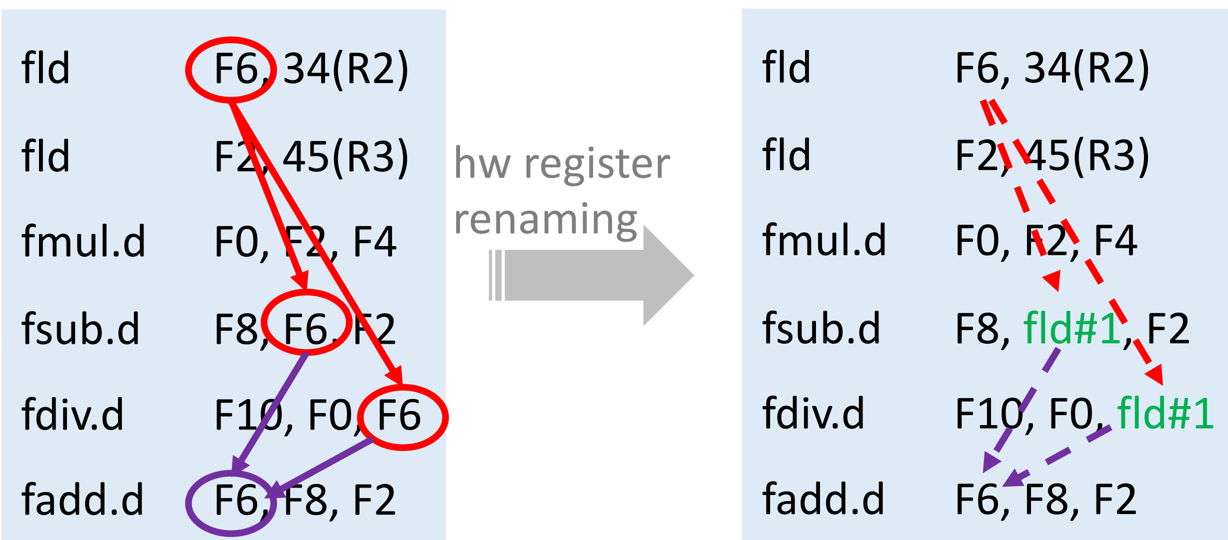
  - Register values are passed through the register file; a child instruction starts execution after the last parent finishes execution
  - Pipeline stalls if any name dependence (WAR or WAW) is detected

How? Just recognize the false dependencies as a hazard and stall.

http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/SSC_course_5_Scoreboard_ex.pdf

Tomasulo Algorithm

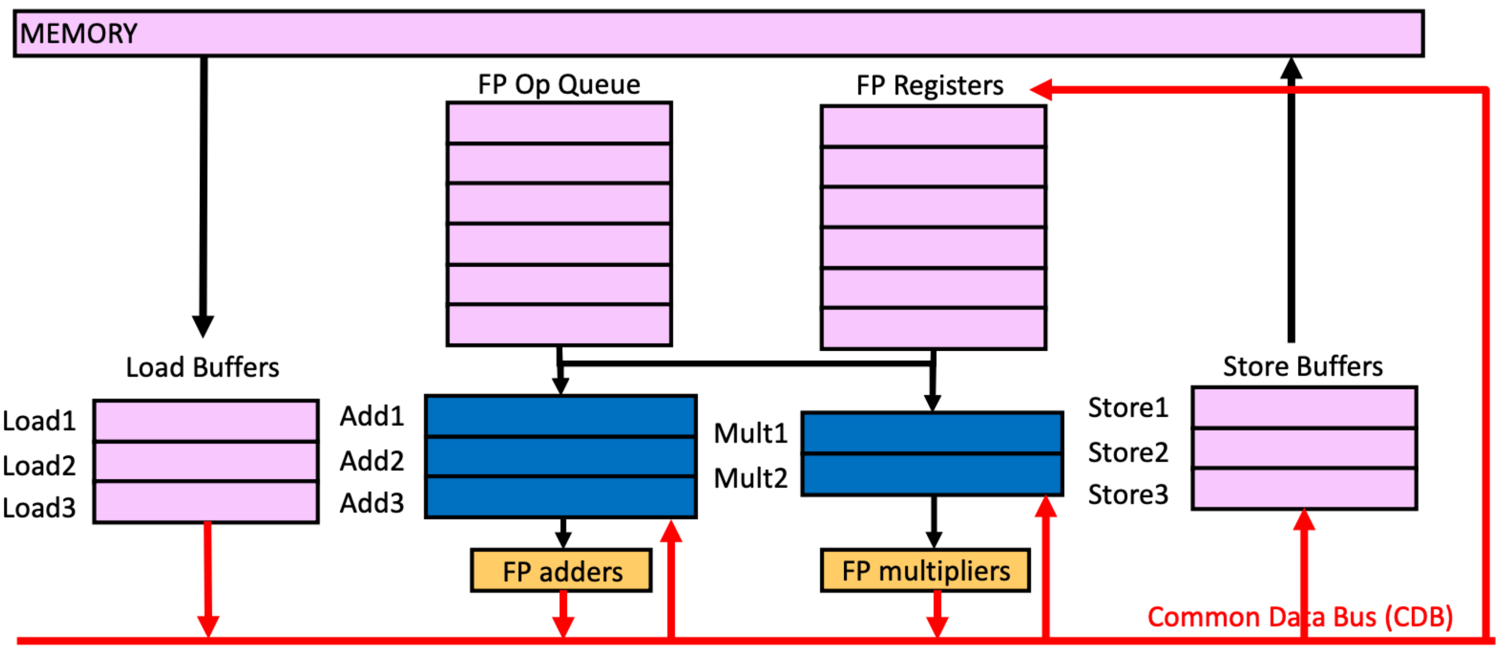
- Key idea: remove dependencies with..
  - 1) HW register renaming
    - What compiler cannot do
  - 2) Data forwarding



http://camelab.org/uploads/Main/lecture09-Tomasulo.pdf

Tomasulo Organization

- Control & buffers are distributed with Function Units (FU)
  - FU buffers called “Reservation Stations (RS)”; have pending ops
  - Registers in instructions replaced by values or pointers to RS
- Load and Store treated as FUs with RSs as well
- Results to FU from RS, not through registers, over Common Data Bus (CDB) that broadcasts results to all FUs

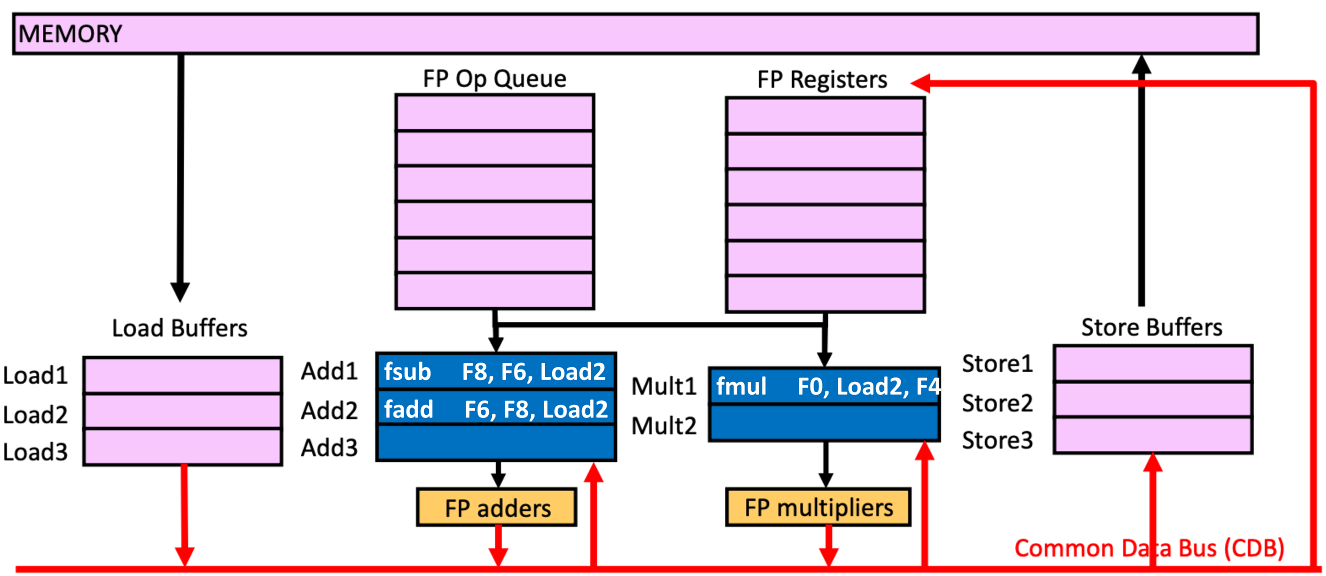


http://camelab.org/uploads/Main/lecture09-Tomasulo.pdf

Three Stages of Tomasulo

- Stage-1: **Issue**
- Get an instruction from FP Op Queue
  - If the reservation station is free (no structural hazard), the control issues such instruction and sends corresponding operands (renames registers)
    - Register are renamed in this step, eliminating WAR and WAW hazards

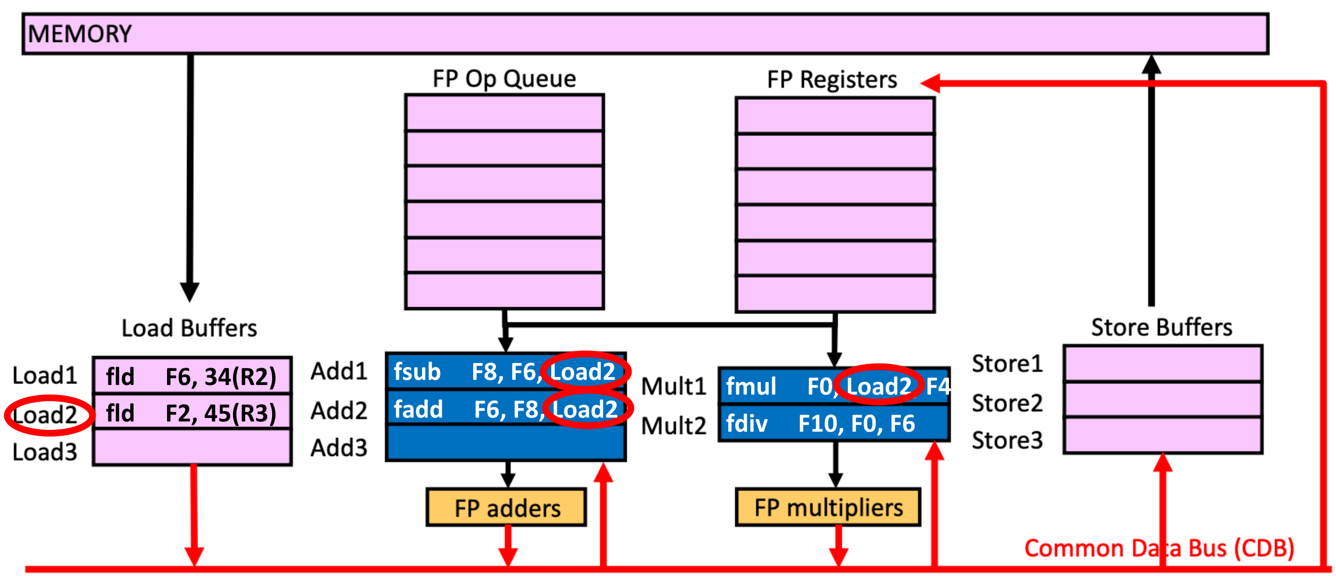
fld	F6, 34(R2)
fld	F2, 45(R3)
fmul.d	F0, F2, F4
fsub.d	F8, F6, F2
fdiv.d	F10, F0, F6
fadd.d	F6, F8, F2



Three Stages of Tomasulo (cont.)

- Stage-2: **Execute**
- Operate on operands (EX)
  - When both operands are ready, it executes; otherwise, it checks up the CDB for results
    - Instructions are delayed here until all of their operands are available, eliminating RAW hazards

fld	F6, 34(R2)
fld	F2, 45(R3)
fmul.d	F0, F2, F4
fsub.d	F8, F6, F2
fdiv.d	F10, F0, F6
fadd.d	F6, F8, F2

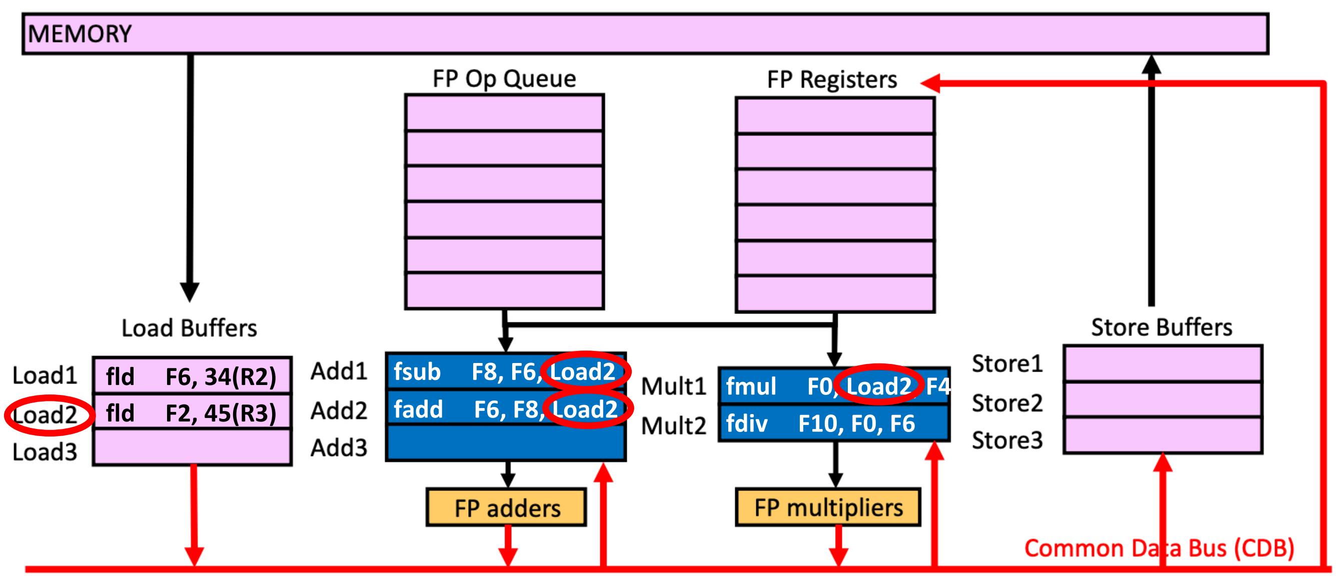


http://camelab.org/uploads/Main/lecture09-Tomasulo.pdf

Three Stages of Tomasulo (cont.)

- Stage-3: **Write result**
- Finish execution:
  - ALU operations results are written back to registers and store operations are written back to memory
    - If the result is available, write it on the CDB and from there into the registers and any reservation stations waiting for this result

fld	F6, 34(R2)
fld	F2, 45(R3)
fmul.d	F0, F2, F4
fsub.d	F8, F6, F2
fdiv.d	F10, F0, F6
fadd.d	F6, F8, F2



http://camelab.org/uploads/Main/lecture09-Tomasulo.pdf

Simple Tomasulo Data Structures

- Three main components
  - Instruction status
  - Reservation stations (Load buffer & FU buffer)
    - Scheduling: waiting operands
    - Register renaming: remove false dep.
  - Register result status

Scoreboard

Insn Status							
Inst	dst	src1	src2	D	S	X	W
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

FU Status									
FU	B	Op	dst	src1	src2	Q1	Q2	R1	R2
Int									
Mult1									
Mult2									
Add									
Div									

Reg Status	
	FU
F0	
F2	
F4	
F6	
F8	
F10	
...	

Tomasulo

Insn Status						
Inst	dst	src1	src2	D	X	W
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

RS (Load buffer)		
	B	Addr
LD1		
LD2		
LD3		

Reservation Stations (FU buffer)						
FU	B	Op	V1	V2	Q1	Q2
Add1						
Add2						
Add3						
Mult1						
Mult2						

Reg Status	
	FU
F0	
F2	
F4	
F6	
F8	
F10	
...	

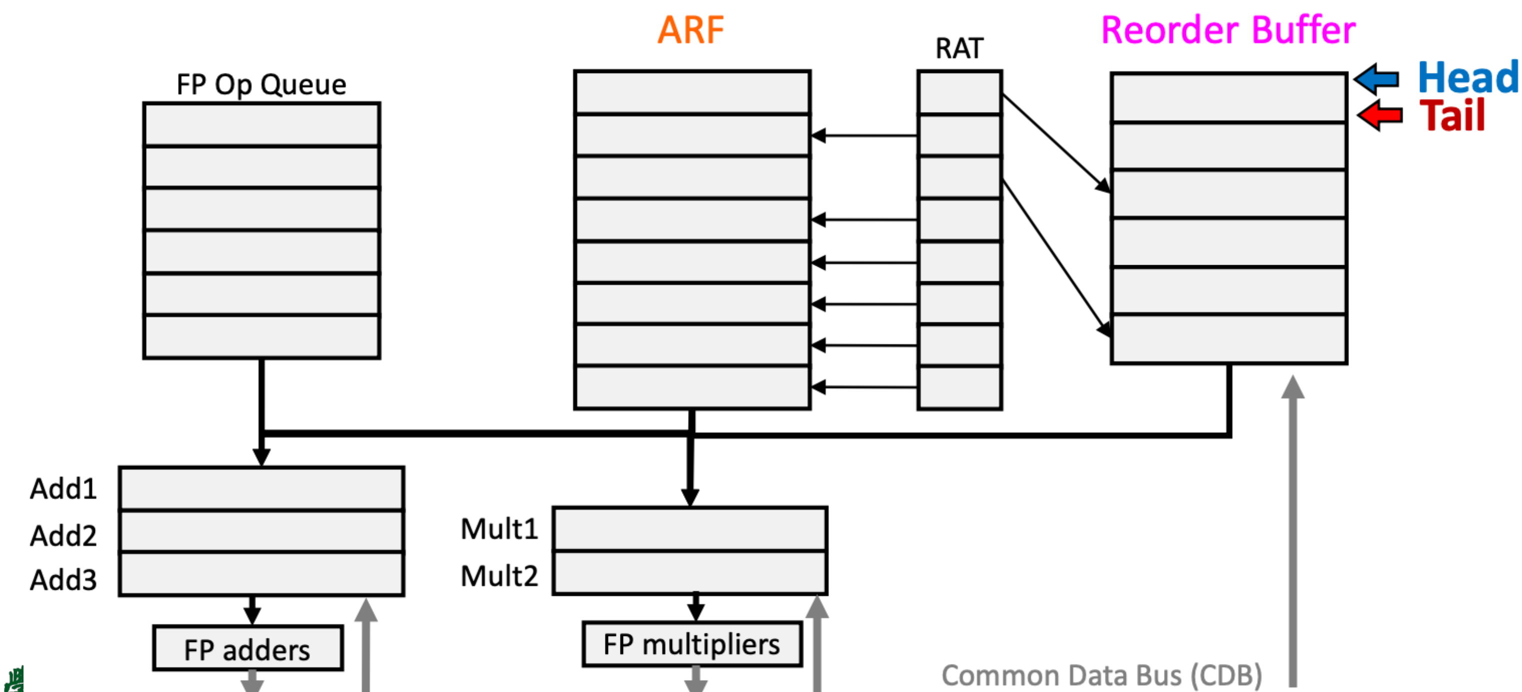
http://camelab.org/uploads/Main/lecture09-Tomasulo.pdf

Reorder Buffer[重排序缓存]

- In the Tomasulo architecture, instructions complete in an *out-of-order*
  - Exceptions are non-trivial to handle
  - Branch misprediction is also difficult to recover from
- **Reorder Buffer** (ROB) enables to finish instructions in the program order
  - And, allows to free RS earlier
  - ROB holds the result of inst between completion and commit
- Key idea of ROB: execute the insts in out of program order, but make outside world can “believe” it’s in-order
  - Solution: Re-Order Buffer+ Architected Register File
    - ROB: keep the temporal results (executed in out-of-order)
    - ARF: keep the final results (illusion of in-order execution)

Tomasulo w/ ROB Organization

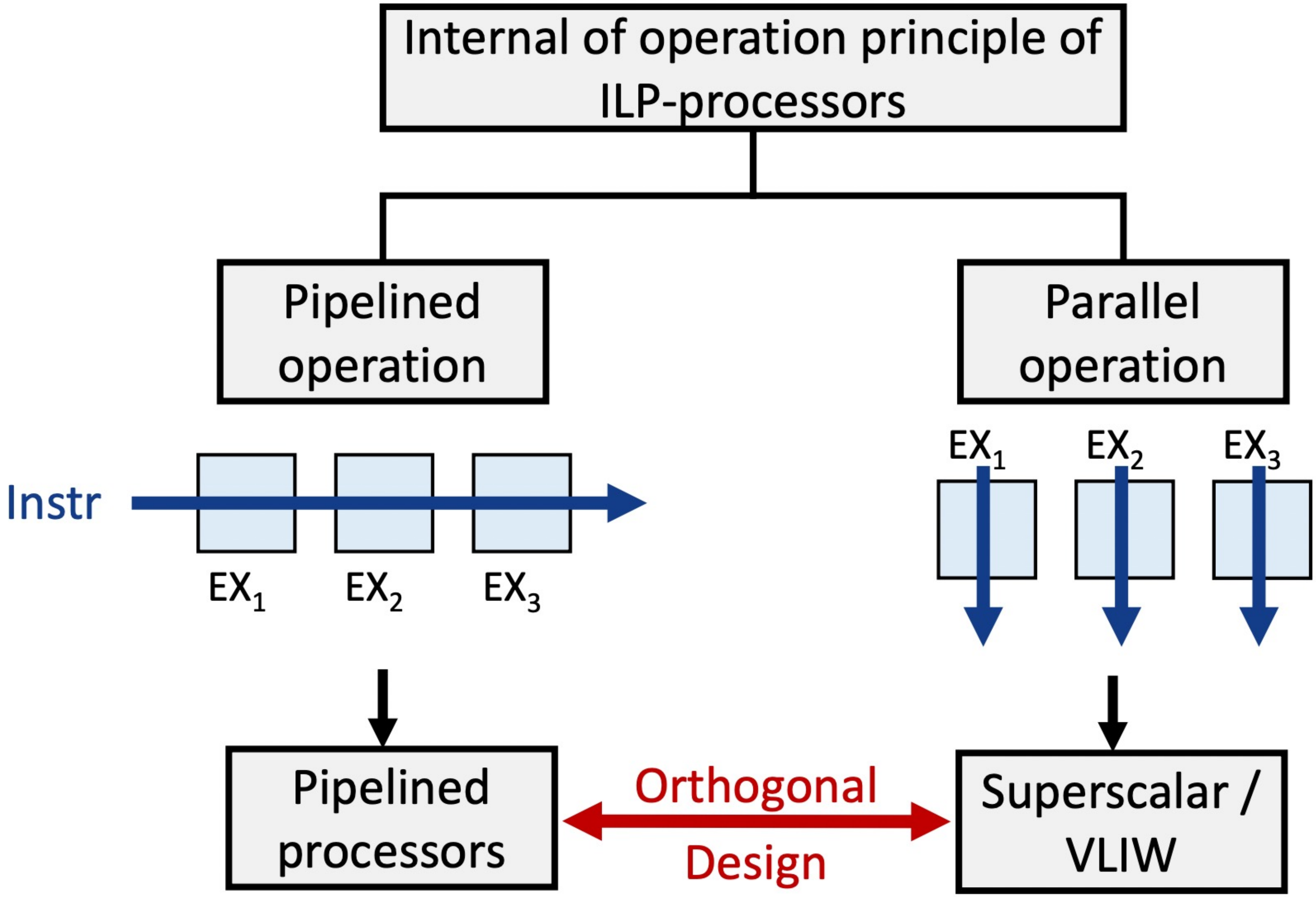
- Re-Order buffer is based on Tomasulo
- Just renamed FP register to ARF (Architected Register File)
- Add Re-Order buffer for out-of-order results
  - Buffer is managed with two pointers (head & tail)
- RAT (Register Alias Table) keeps the register renaming info



Reorder Buffer Procedure[过程]

- Issue
  - Allocate reservation station(RS) and Reorder Buffer(ROB), read available operands
- Execute
  - Begin execution when operand values are available
- Write Result
  - Write result and ROB tag on CDB
- Commit
  - When ROB reaches head, update register
  - When a mispredicted branch reaches head of ROB, discard all entries

Another ILP



http://camelab.org/uploads/Main/lecture06-istruction-paralllel-processing.pdf

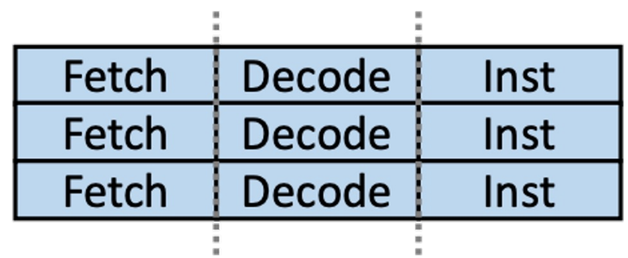
Multiple Issue[多发射]

- To achieve CPI < 1, need to complete multiple instructions per clock
- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors

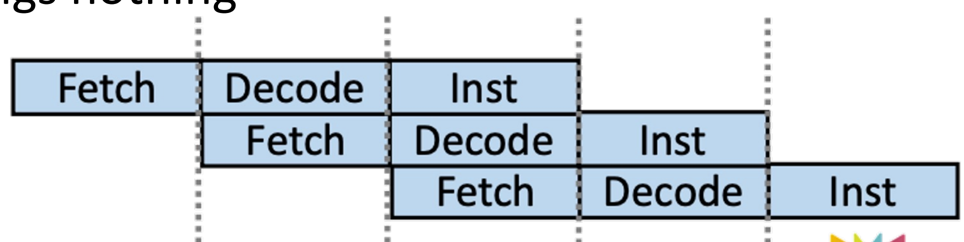
Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Superscalar[超标量]

- Superscalar architectures allow several instructions to be issued and completed per clock cycle
- A superscalar architecture consists of a number of pipelines that are working in parallel (N-way Superscalar)
  - Can issue up to N instructions per cycle
- Superscalarity is Important
  - Ideal case of N-way Super-scalar
    - All instructions were independent
    - Speedup is “N” (Superscalarity)



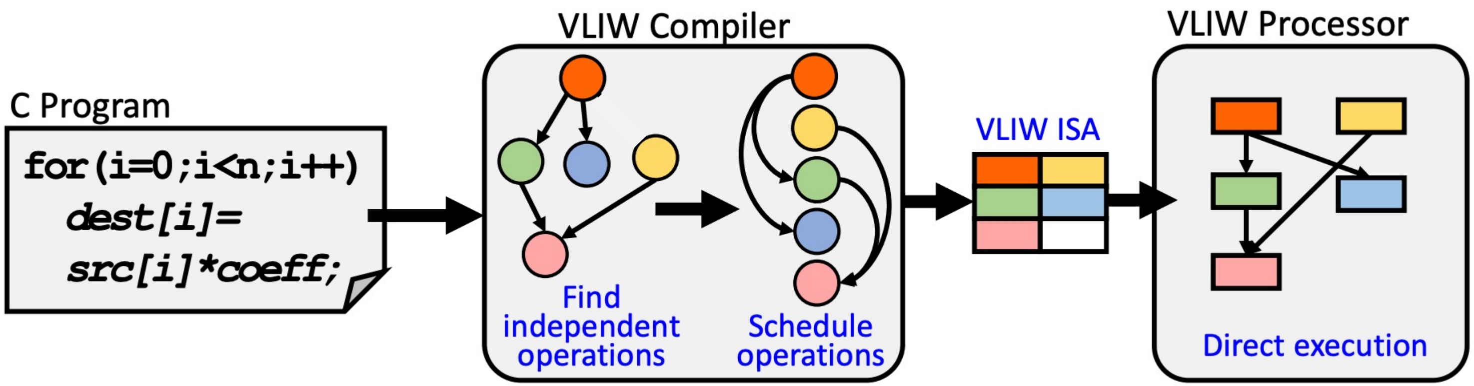
  - What if all instructions are dependent?
    - No speed up, super-scalar brings nothing
    - (Just similar to pipelining)



http://camelab.org/uploads/Main/lecture06-istruction-paralllel-processing.pdf

VLIW Processor[超长指令字]

- Static multiple-issue processors (decision making at compile time by the compiler)
  - Package multiple operations into one instruction
- Key idea: replace a traditional sequential ISA with a new ISA that enables the compiler to encode ILP directly in the hw/sw interface
  - Sub-instructions within a long instruction must be independent
  - Multiple "sub-instructions" packed into one long instruction
  - Each "slot" in a VLIW instruction for a specific functional unit

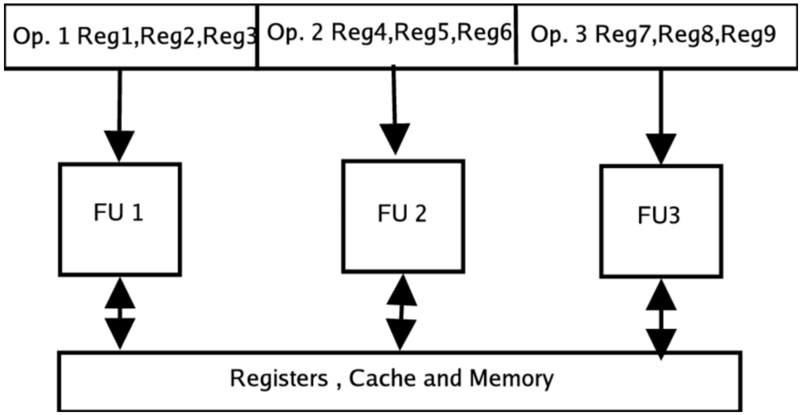


http://camelab.org/uploads/Main/lecture06-istruction-paralllel-processing.pdf

VLIW Processor (cont.)

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility



Summary: Tomasulo

- To support dynamic scheduling
  - Dynamically determining when an inst is ready to execute
  - Avoid unnecessary hazards
    - RAW hazards: avoided by executing an inst only when its operands are available
    - WAR and WAW hazards: eliminated by register renaming
  - Register renaming is provided by reservation stations
- To support speculation
  - Speculate the branch outcome and execute as if guesses are correct
  - Allow insts execute OoO but to force them to commit in order
  - Reorder buffer: hold the results of insts that have finished execution but have not committed
    - Pass results among insts that may be speculated

Summary: Multiple Issue

- Single issue: ideal CPI of one
  - Issue only one inst every clock cycle
  - Techniques to eliminate data, control stalls
- Multiple issue: ideal CPI less than one
  - Issue multiple insts in a clock cycle
  - **Statically scheduled superscalar** processors
    - Issue varying number of insts per clock, execute in-order
  - **VLIW** (very long inst word) processors
    - Issue a fixed number of insts formatted as one large inst
    - Inherently statically scheduled by the compiler
  - **Dynamically scheduled superscalar** processors
    - Issue varying number of insts per clock, execute OoO

Computer Architecture

计 算 机 体 系 结 构

第7讲：DLP & GPU（1）

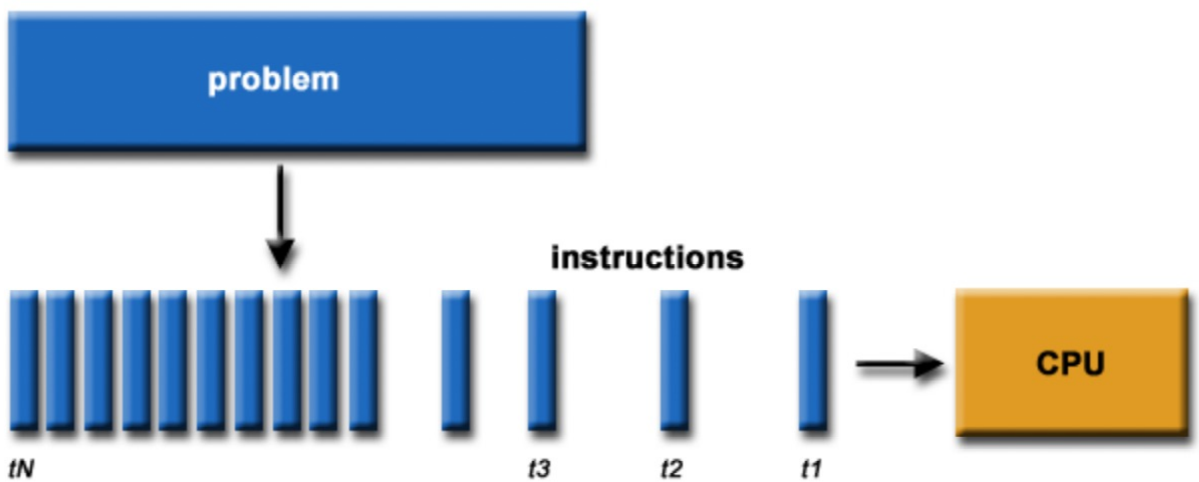
张献伟

xianweiz.github.io

DCS3013, 10/26/2022

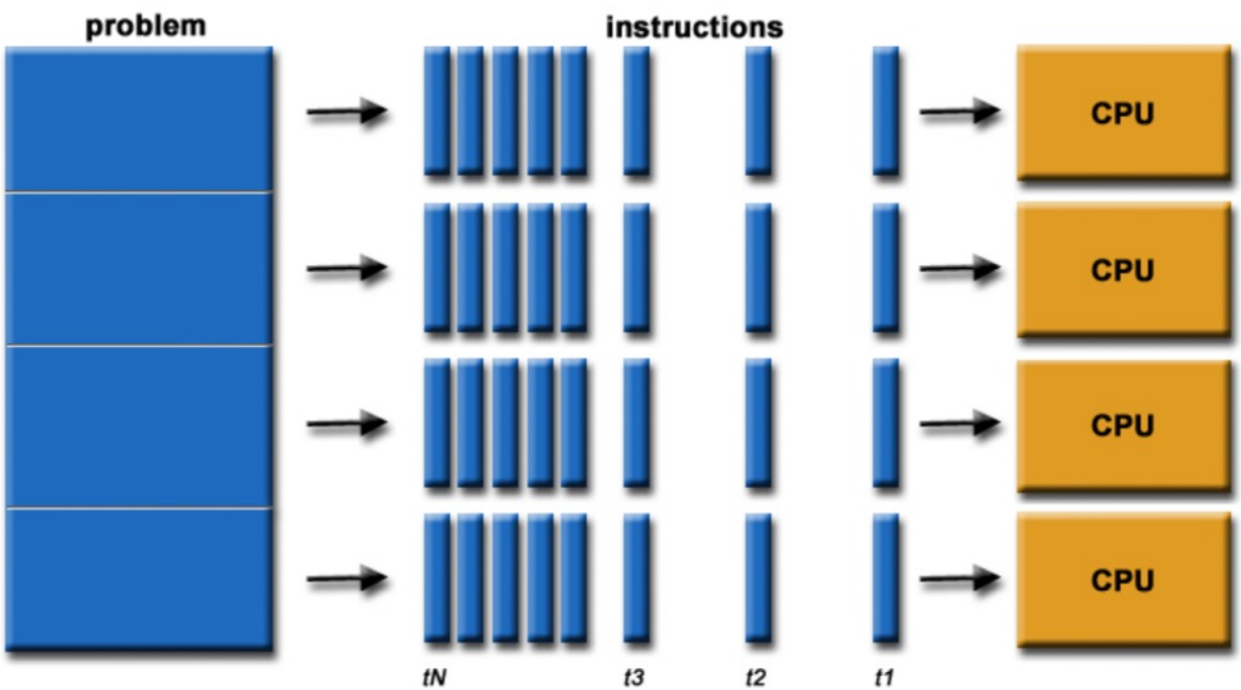
Serial Computing[串行计算]

- Traditionally, software has been written for serial computation
  - To be run on a single computer having a single CPU
  - A problem is broken into a discrete series of instructions
  - Instructions are executed one after another
  - Only one instruction may execute at any moment



Parallel Computing[并行计算]

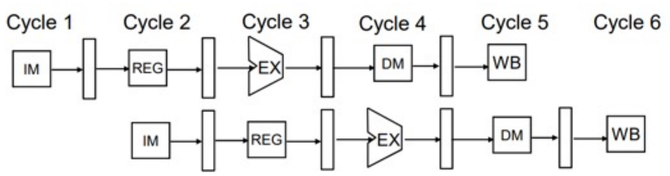
- Simultaneously use multiple compute resources to solve a computational problem
  - Typically in high-performance computing (HPC)
- HPC focuses on performance
  - To solve biggest possible problems in the least possible time



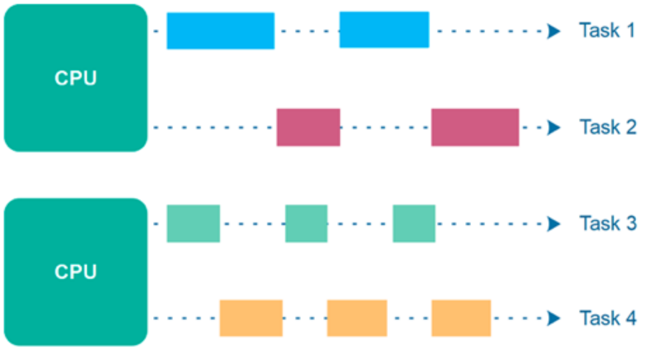
https://www.ima.umn.edu/materials/2010-2011/T11.28-29.10/10287/IMA-PPtTutorial.pdf

Types of Parallel Computing[并行类型]

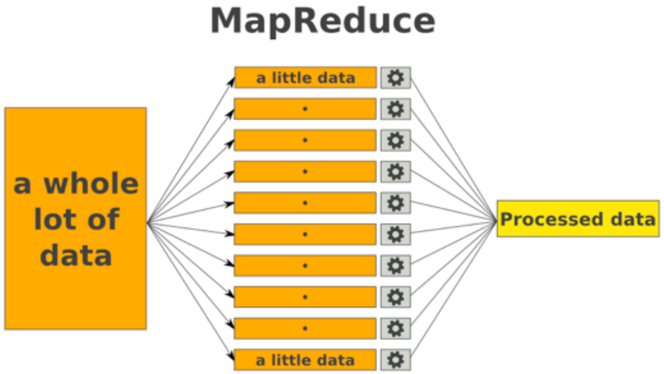
- Instruction level parallelism[指令级并行]
  - Classic RISC pipeline (fetch, ..., write back)



- Task parallelism[任务级并行]
  - Different operations are performed concurrently
  - Task parallelism is achieved when the processors execute on the same or different data



- Data parallelism[数据级并行]
  - Distribution of data across different parallel computing nodes
  - Data parallelism is achieved when each processor performs the same task on different pieces of the data



https://www.uio.no/studier/emner/matnat/ifi/IN5050/v20/undervisningsmaterialet/in5050-simd.pdf

Taxonomy[分类]

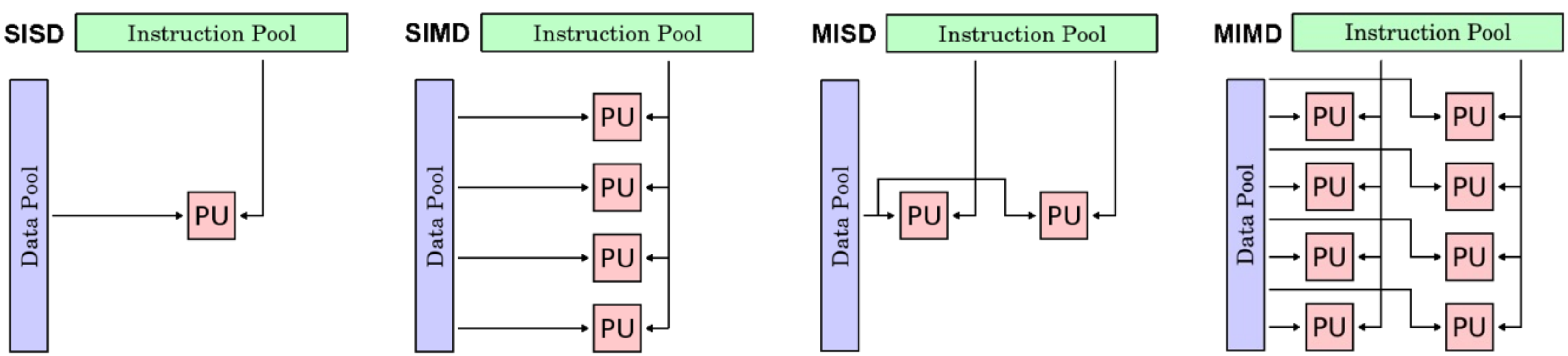
- **Flynn's Taxonomy** (1966) is widely used to classify parallel computers
  - Distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction Stream*** and ***Data Stream***
  - Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***
- 4 possible classifications according to Flynn

SISD Single Instruction stream Single Data stream	**SIMD** Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	**MIMD** Multiple Instruction stream Multiple Data stream

https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

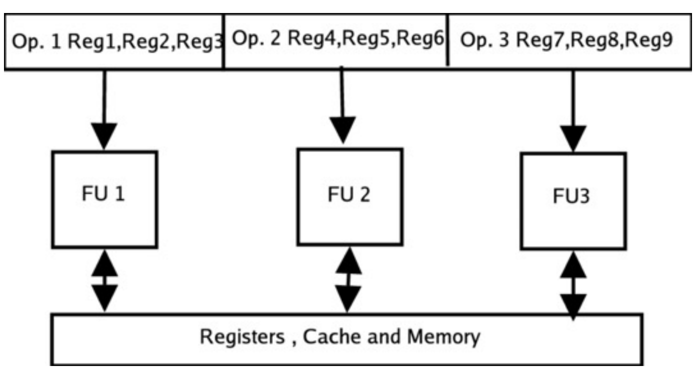
Taxonomy (cont.)

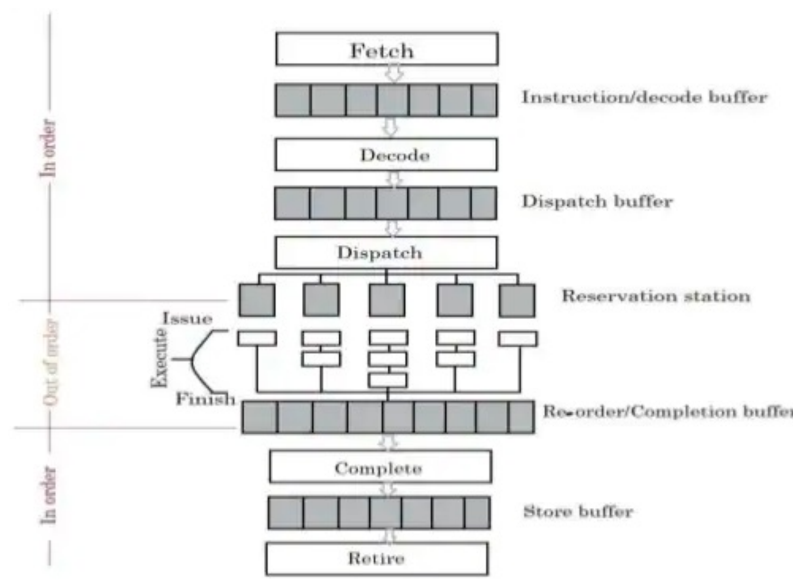
- SISD: single instruction, single data
  - A serial (non-parallel) computer
- **SIMD**: single instruction, multiple data
  - Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing
- MISD: multiple instruction, single data
  - Few (if any) actual examples of this class have ever existed
- MIMD: multiple instruction, multiple data
  - Examples: supercomputers, multi-core PCs, VLIW



SIMD: vs. superscalar and VLIW[对比]

- SIMD performs the same operation on multiple data elements with one single instruction
  - Data-level parallelism
- Superscalar dynamically issues multi insts per clock[超标量]
  - Instruction level parallelism (ILP)
- VLIW receives long instruction words, each comprising a field (or opcode) for each execution unit[超长指令字]
  - Instruction level parallelism (ILP)

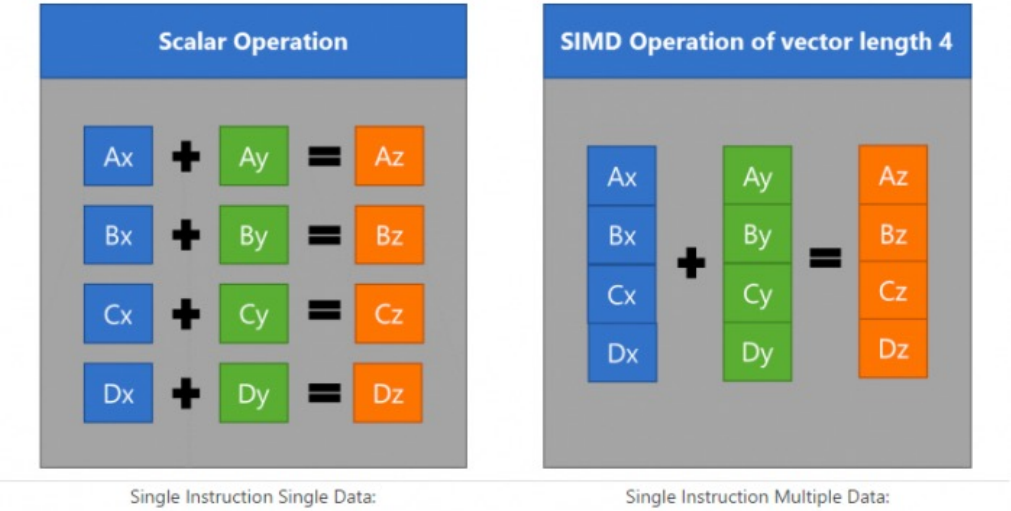




中山大學 SUN YAT-SEN UNIVERSITY

SIMD: Vector Processors[向量处理器]

- Vector processor (or array processor)[处理器]
  - CPU that implements an instruction set containing instructions that operate on one-dimensional arrays (vectors)
- People use vector processing in many areas[应用]
  - Scientific computing
  - Multimedia processing (compression, graphics, image processing, …)
- Instruction sets[指令集]
  - MMX
  - SSE
  - AVX
  - NEON
  - …



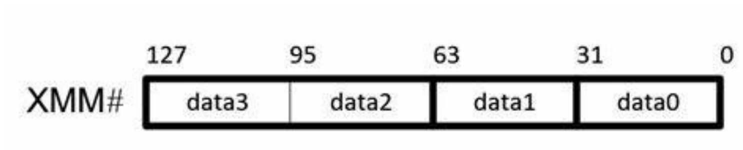
https://www.uio.no/studier/emner/matnat/ifi/IN5050/v20/undervisningsmaterialet/in5050-simd.pdf

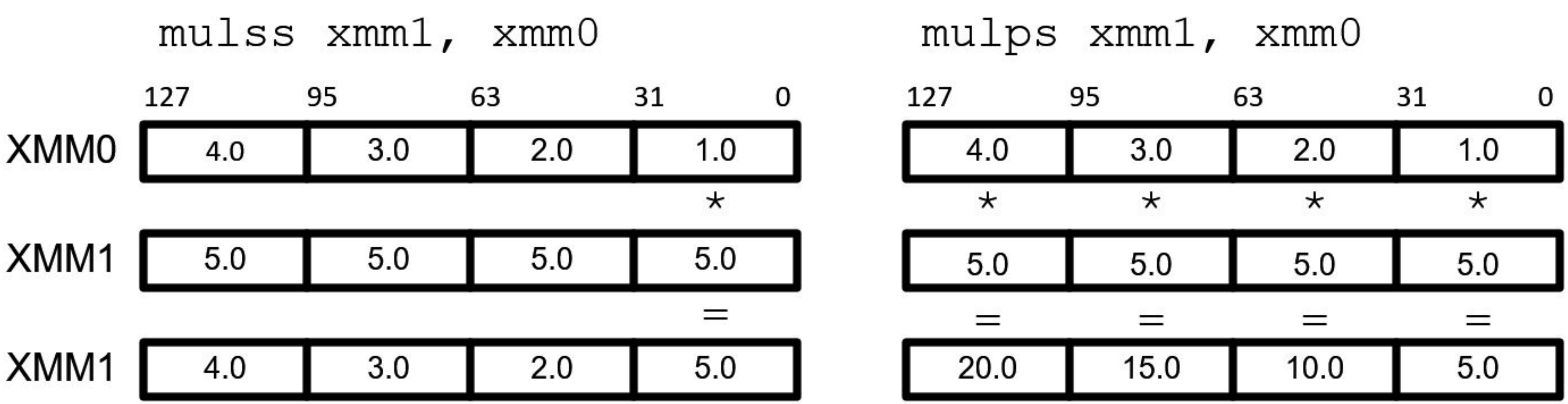
SIMD: MMX

- MMX is officially a meaningless initialism trademarked by Intel; unofficially,
  - MultiMedia eXtension
  - Multiple Math eXtension
  - Matrix Math eXtension
- Introduced on the "Pentium with MMX Technology" in 1998
- SIMD computation processes multiple data in parallel with a single instruction
  - MMX gives 2 x 32-bit computations at once
  - MMX defined 8 "new" 64-bit integer registers (mm0 ~ mm7)
  - 3DNow! was the AMD extension of MMX

SIMD: SSE

- Streaming SIMD Extensions
  - SSE defines 8 new 128-bit registers (xmm0 ~ xmm7) for FP32 computations
    - Since each register is 128-bit long, we can store total 4 FP32 numbers
  - 4 simultaneous 32-bit computations

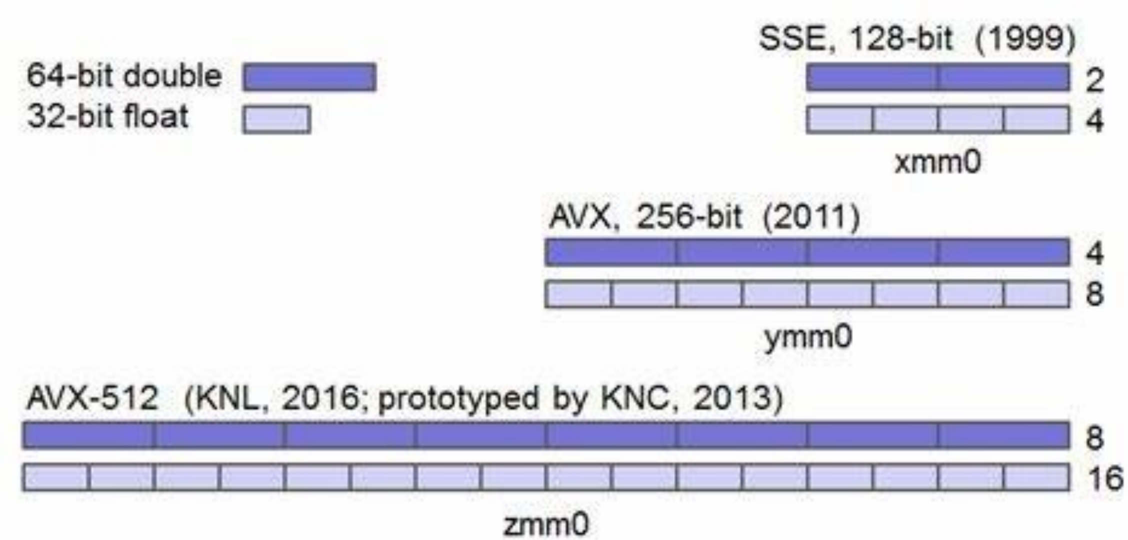




https://www.uio.no/studier/emner/matnat/ifi/IN5050/v20/undervisningsmaterialet/in5050-simd.pdf

SIMD: AVX

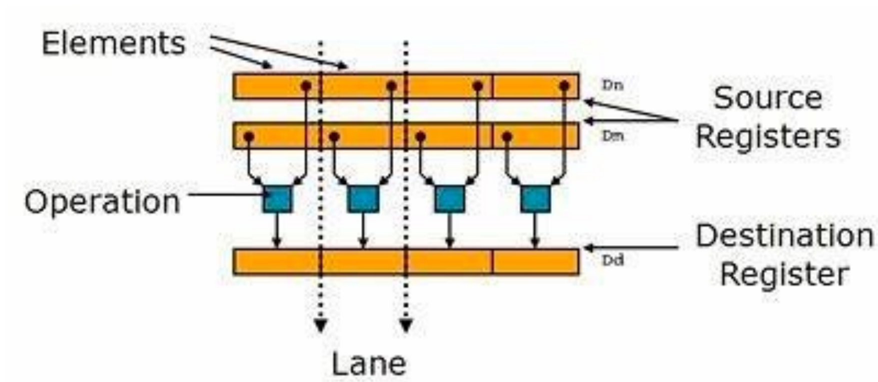
- Advanced Vector Extensions (AVX)
  - A new-256 bit instruction set extension to SSE
    - 16-registers available in x86-64
    - Registers renamed from XMMi to YMMi
  - Yet a proposed extension is AVX-512
    - A 512-bit extension to the 256-bit XMM
    - Supported in from Intel's Xeon Phi x200 (Knights Landing) and Skylake-SP, and onwards



https://www.uio.no/studier/emner/matnat/ifi/IN5050/v20/undervisningsmaterialet/in5050-simd.pdf

SIMD: NEON

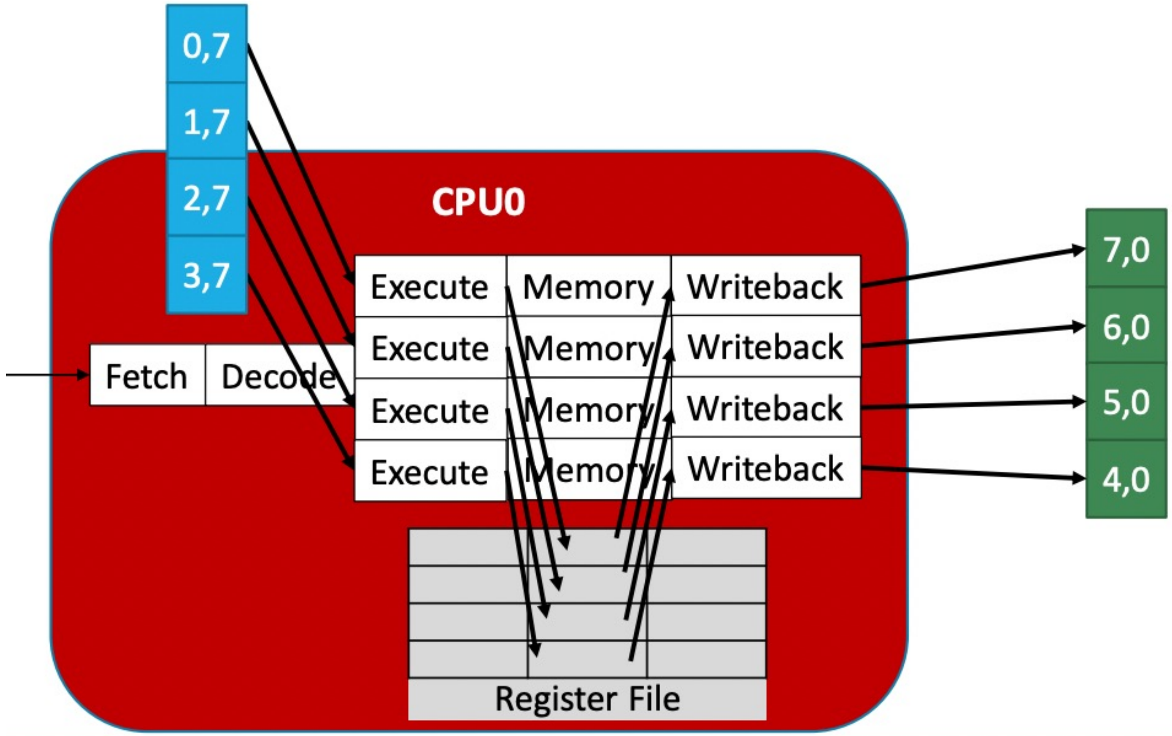
- ARM Advanced SIMD Extensions
  - Introduced by ARM in 2004 to accelerate media and signal processing
    - NEON can for example execute MP3 decoding on CPUs running at 10 MHz
  - 128-bit SIMD Extension for the ARMv7 & ARMv8
    - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit or 64-bit



https://www.uio.no/studier/emner/matnat/ifi/IN5050/v20/undervisningsmaterialet/in5050-simd.pdf

Data Parallelism: SIMD

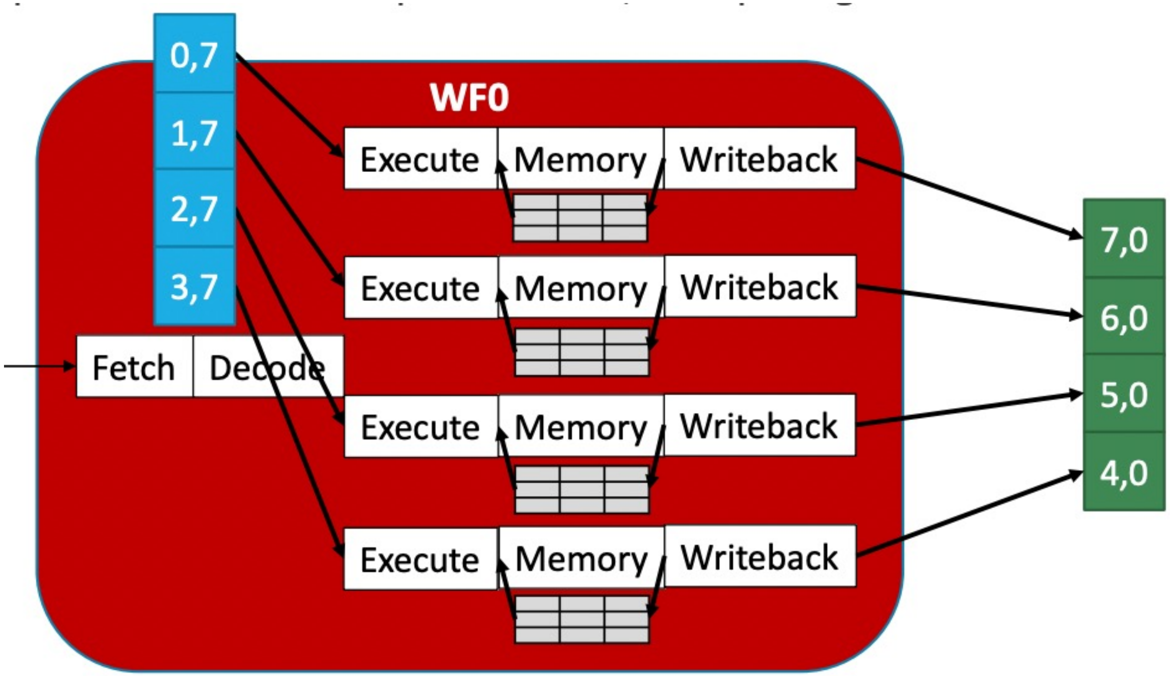
- Single Instruction Multiple Data
  - Split identical, independent work over multiple execution units (lanes)
  - More efficient: eliminate redundant fetch/decode
  - One Thread + Data Parallel Ops → Single PC, single register file



https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf

Data Parallelism: SIMT

- Single Instruction Multiple Thread
  - Split identical, independent work over multiple threads
  - Multiple Threads + Scalar Ops → One PC, multiple register files
  - ≈ SIMD + multithreading
  - Each thread has its own registers



https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf

Execution Model[执行模型]

MIMD

SIMD

SIMT

Multiple independent threads

Multicore CPUs

One thread with wide execution datapath

x86 SSE/AVX

Multiple lockstep threads

GPUs

- SI(MD/MT)
  - Broadcasting the same instruction to multiple execution units
  - Replicate the execution units, but they all share the same fetch/decode hardware

SIMD and SIMT are used interchangeably

https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf

SIMD: GPU vs. CPU/Traditional

- Traditional SIMD contains a single thread
  - Programming model is SIMD (no threads)
  - SW needs to know vector length
  - ISA contains vector/SIMD instructions
- GPU SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
- Essentially, it is SPMD programming model implemented on SIMD hardware

中山大學 SUN YAT-SEN UNIVERSITY

https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module5.1.3-simd-and-gpus-part3-vliw-dae-systolic.pdf

NSCC GZ

Example: add two vectors

C:

```
for(i=0;i<n;++i) a[i]=b[i]+c[i];
```

Matlab:

```
a=b+c;
```

SIMD:

```
void add(uint32_t *a, uint32_t *b, uint32_t *c, int n) {
    for(int i=0; i<n; i+=4) {
        //compute c[i], c[i+1], c[i+2], c[i+3]
        uint32x4_t a4 = vld1q_u32(a+i);
        uint32x4_t b4 = vld1q_u32(b+i);
        uint32x4_t c4 = vaddq_u32(a4,b4);
        vst1q_u32(c+i,c4);
    }
}
```

SIMT:

```
__global__ void add(float *a, float *b, float *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    a[i]=b[i]+c[i]; //no loop!
}
```