Advanced Computer Architectures

05

Superscalar techniques - Memory data flow, VLIW and EPIC processors

Czech Technical University in Prague, Faculty of Electrical Engineering
Slides authors: Michal Štepanovský, update Pavel Píša

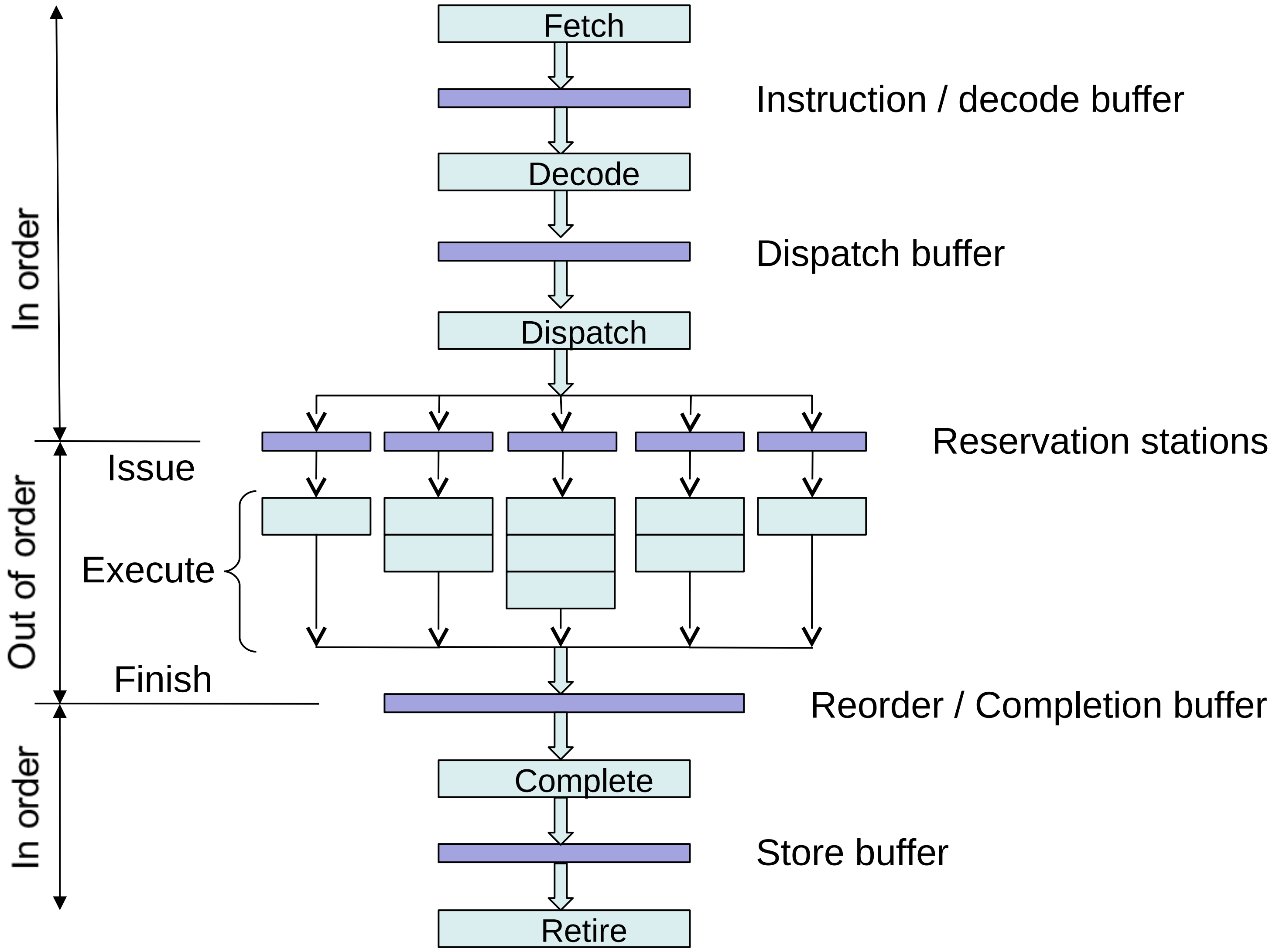
Superscalar Techniques…

- The goal is to achieve a **maximum throughput of instruction processing**
- Instruction processing can be analyzed as instructions flow or data flow, more precisely:
  - register data flow – data flow between processor registers
  - instruction flow through the pipeline
  - **memory data flow** – to/from memory

Today's lecture topic

- It roughly matches to:
  - Arithmetic-logic (ALU) and other computational instructions (FP, bit-field, vector) processing
  - Branch instruction processing
  - Load/store instruction processing
- maximizing the throughput of these three flows (or complete flow) correspond to the minimizing penalties and latencies of above three instructions types

Superscalar pipeline – see previous lesson



What you know already from other subjects ...

- **Load / Store instructions** are responsible for moving data between the memory and the processor's own registers
- The processor has a very limited number of registers
- The compiler generates a so-called ***spill code*** that temporarily saves data to memory/stack to make registers available - just by using the load/store instructions

Significant latency of load/store instructions caused by:

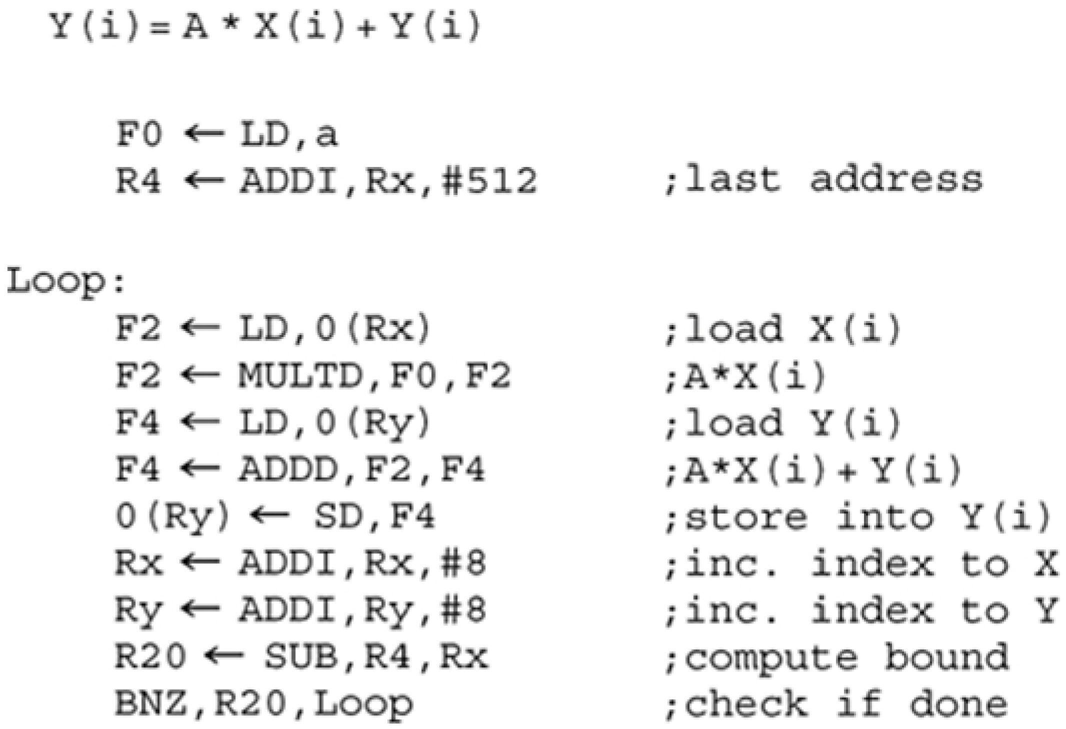
- **Address generation** – effective address computation
- **Address translation** (see virtual memory) – TLB hit vs. *TLB miss* (is the *page* in memory × page fault, is the address in TLB × page-table walking)
- **Actual memory access** – see next slide

Memory access

- Load instruction:
  - Store data read from memory to **rename register** or **reorder buffer**. Instruction finishes execution just at this moment. Architectural registers update waits for instruction completion – completed, released from reorder buffer
- Store instruction:
  - The instruction finishes execution as early as the address is successfully translated. Data (from the register to store) are held in reorder buffer. Actual write is processed after completion of the instruction, not earlier. Why is it that way?
  - **store buffer** - FIFO ; instruction is retired when memory is actualized, Retiring – when a bus is available..

The ordering of the load and store instructions

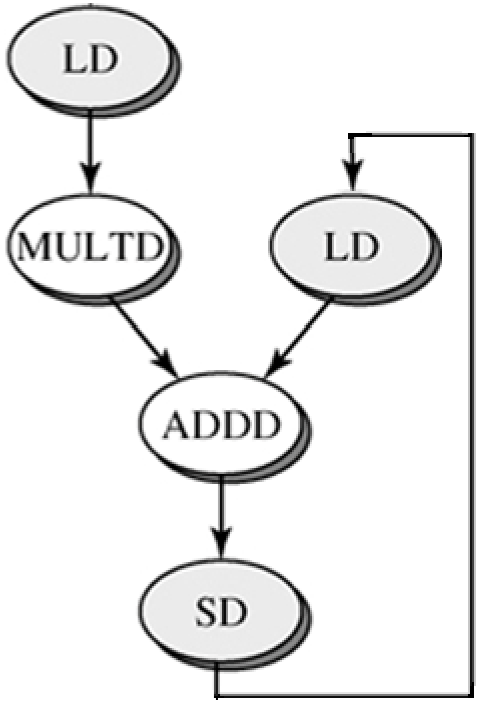
- Data dependencies – RAW, WAR, WAW – between load/store instructions operating with the same address
- **Total ordering** – obey program order of all load/store instructions. Is it necessary?



```
Y(i)=A*X(i)+Y(i)

      F0 ← LD,a
      R4 ← ADDI,Rx,#512      ;last address

  Loop:
      F2 ← LD,0(Rx)          ;load X(i)
      F2 ← MULTD,F0,F2       ;A*X(i)
      F4 ← LD,0(Ry)          ;load Y(i)
      F4 ← ADDD,F2,F4        ;A*X(i)+Y(i)
      0(Ry) ← SD,F4          ;store into Y(i)
      Rx ← ADDI,Rx,#8        ;inc. index to X
      Ry ← ADDI,Ry,#8        ;inc. index to Y
      R20 ← SUB,R4,Rx        ;compute bound
      BNZ,R20,Loop           ;check if done
```



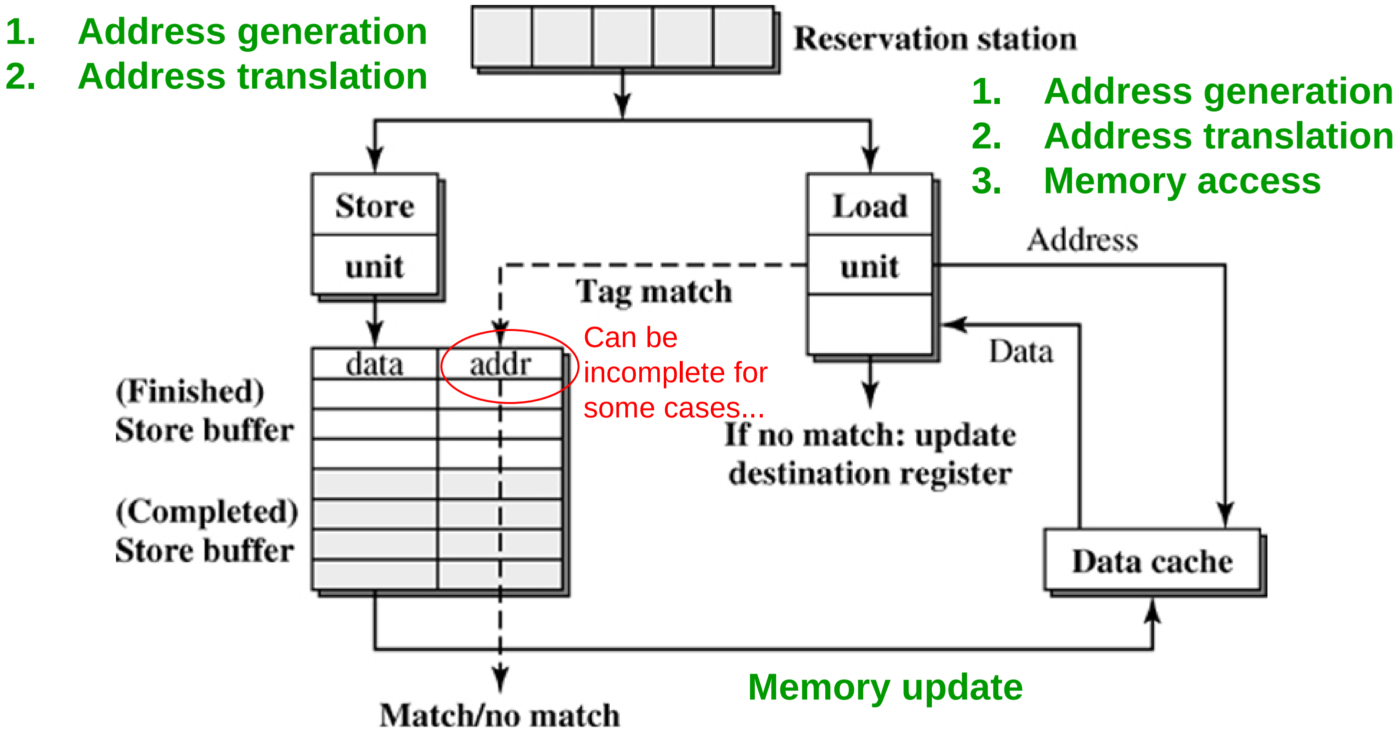
Model of sequential consistency

- **Sequential consistency condition** demands some restrictions to out-of-order load/store instructions executions
- What to do when the exception occurs?
- Memory state has to be equivalent to the **sequential order** of load/store instructions
- This requires that store instructions have to be executed in the program order, or more precisely that memory has to be updated in such sequence as if instruction were executed in the program order
- If **store** instructions are executed in **program order**, the WAW and WAR dependencies are guaranteed. Only RAW dependencies have to solved in a pipeline…
- Load instruction – out-of-order

Load forwarding and Load bypassing

For initial analysis, consider load/store instructions issuing from RS in-order

- ***Load bypassing*** allows execute a load before the store, if instructions are memory independent (does not alias store). Use **Stall** or **Load forwarding** for other cases..

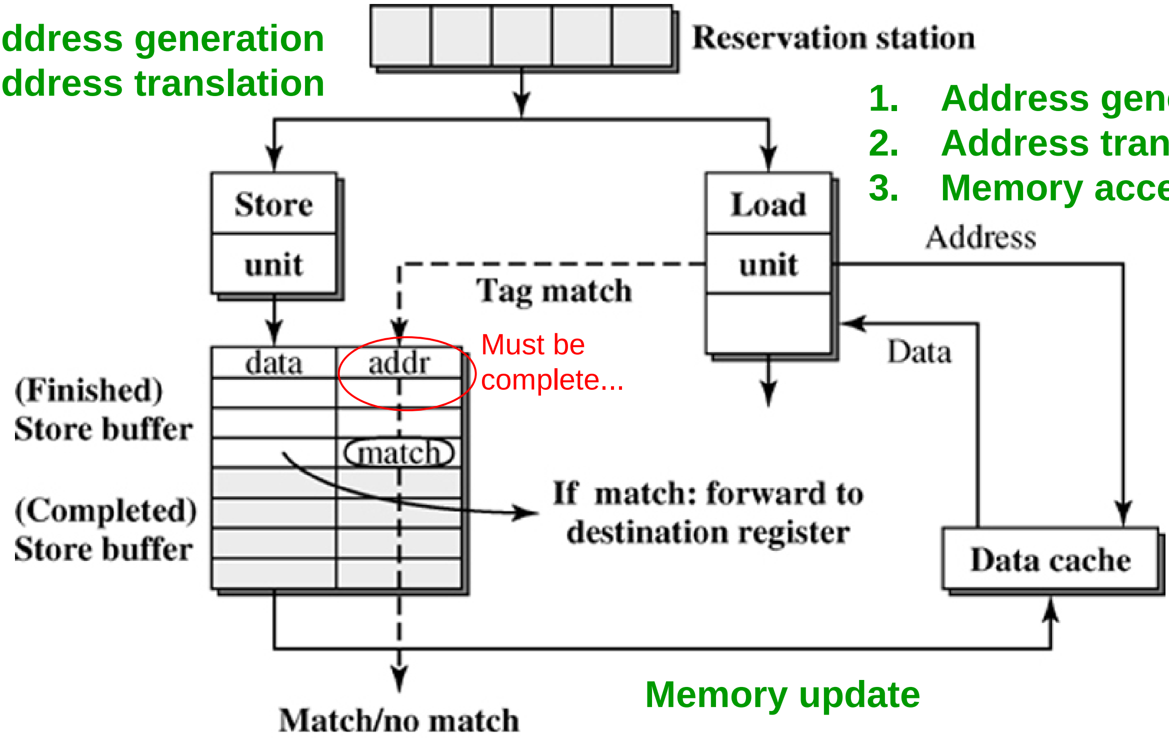


Load forwarding and Load bypassing

For initial analysis, consider load/store instructions issuing from RS in order

- ***Load forwarding*** forwards store instruction data to load instruction to resolve RAW dependencies

1. Address generation
2. Address translation



1. Address generation
2. Address translation
3. Memory access

Must be complete...

Memory update

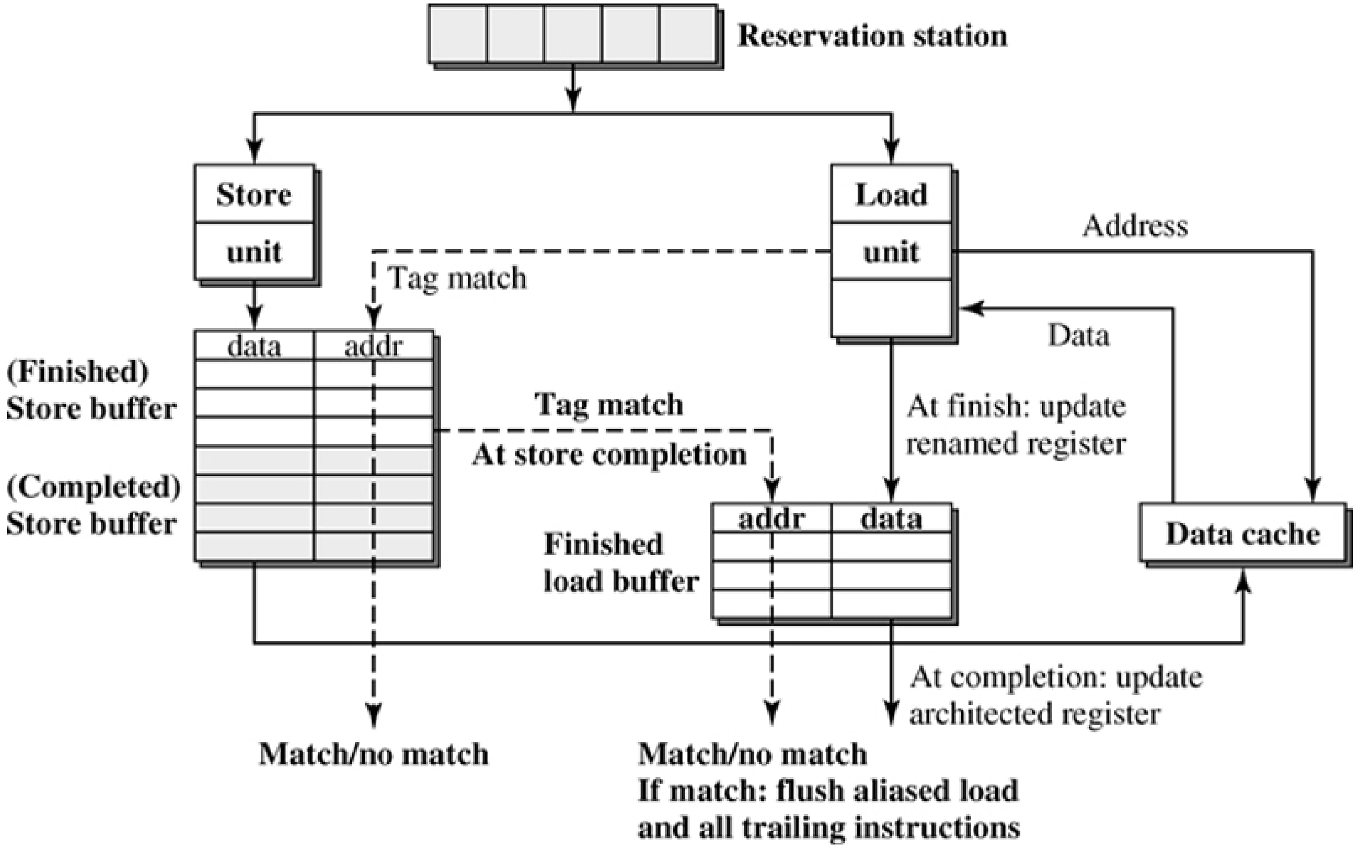
Store: dispatched, issued, finished, completed, retired

Load – if match: do not access or use data from cache but use data from Store buffer

Load forwarding and Load bypassing

- If the instructions from **reservation station are issued out-of-order** then load instruction following store instruction can be executed before preceding store instruction aliasing RAW hazard because the store is not in Store buffer yet (it can be executed, in the reservation station, or even in memory). Even information about its address can be still unknown (RAW dependency cannot be detected).
- Solution?
- It is possible to assume that there is no dependency and correct case it is found later... => **speculative execution**
- Speculative execution is enabled by use of ***Finished load buffer*** (Finish load queue)

Speculative execution of Load instructions



- Load instructions are remembered in **Finished load buffer** until their completion
- Whenever the store reaches completion, alias checking with FLB is processed. No conflict -> store is finished; Conflict -> discard load instr. speculation and reissue it.

Speculative execution

- Load instruction speculations – Why?
- To execute load instructions as early as possible – other computations and instructions depends on it
- Also, earlier load issue detect *cache miss* earlier
- It can compensate the *cache miss penalty* in some cases

- Disadvantages: in the case of incorrect speculation – discard instructions from the load to all following dependencies – it cost time and HW resources which can be better utilized
- That is why to introduce ***Dependence prediction***
  Dependency between store and load is quite predictable for typical programs
- ***Memory dependence predictor*** then decides if the speculative load and following instruction should be started

More instruments to **reduce memory latency**

1. Memory hierarchy: L1 cache, L2, L3…

   Already well known. Do not describe it again.

2. Non-blocking cache, look-up free cache use

   Traditional approach: If the cache miss is detected the load and following loads execution is stalled until data are available.
   Non-blocking cache approach: Load instruction causing miss is put aside (into missed load queue) and continue. Naturally, dependencies on „unserved“ load instruction have to be considered (stall or value prediction->speculation).

3. Prefetching cache

   Future miss is anticipated and cache fill is processed in advance. This requires *memory reference prediction table* and *prefetch queue*.

Non-blocking cache

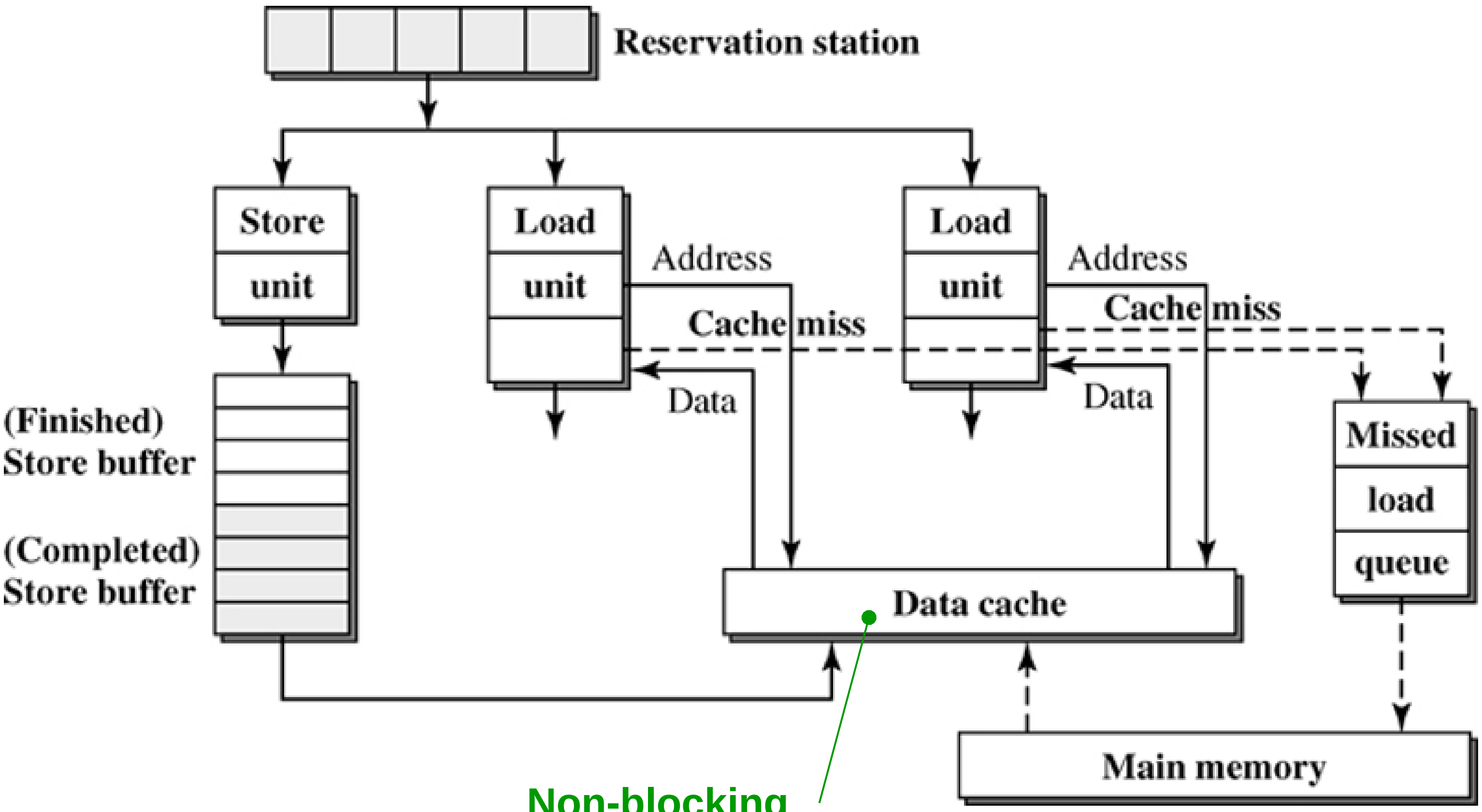
Basic idea:

- Another access (hit) is allowed even that miss is detected and fill is in progress: **hit-under-miss** (the second miss is resolved by stall)
- **miss-under-miss** (or hit-under-multiple-misses) Example: Pentium Pro - 4 unserved memory misses

Cases when the use of non-blocking cache provides advantage:

- When CPU can process more than one load/store in parallel (i.e., superscalar processors)
- If the cache is common for more than one processor (or cache)

Non-blocking cache – Example of superscalar CPU



Non-blocking dual-ported cache

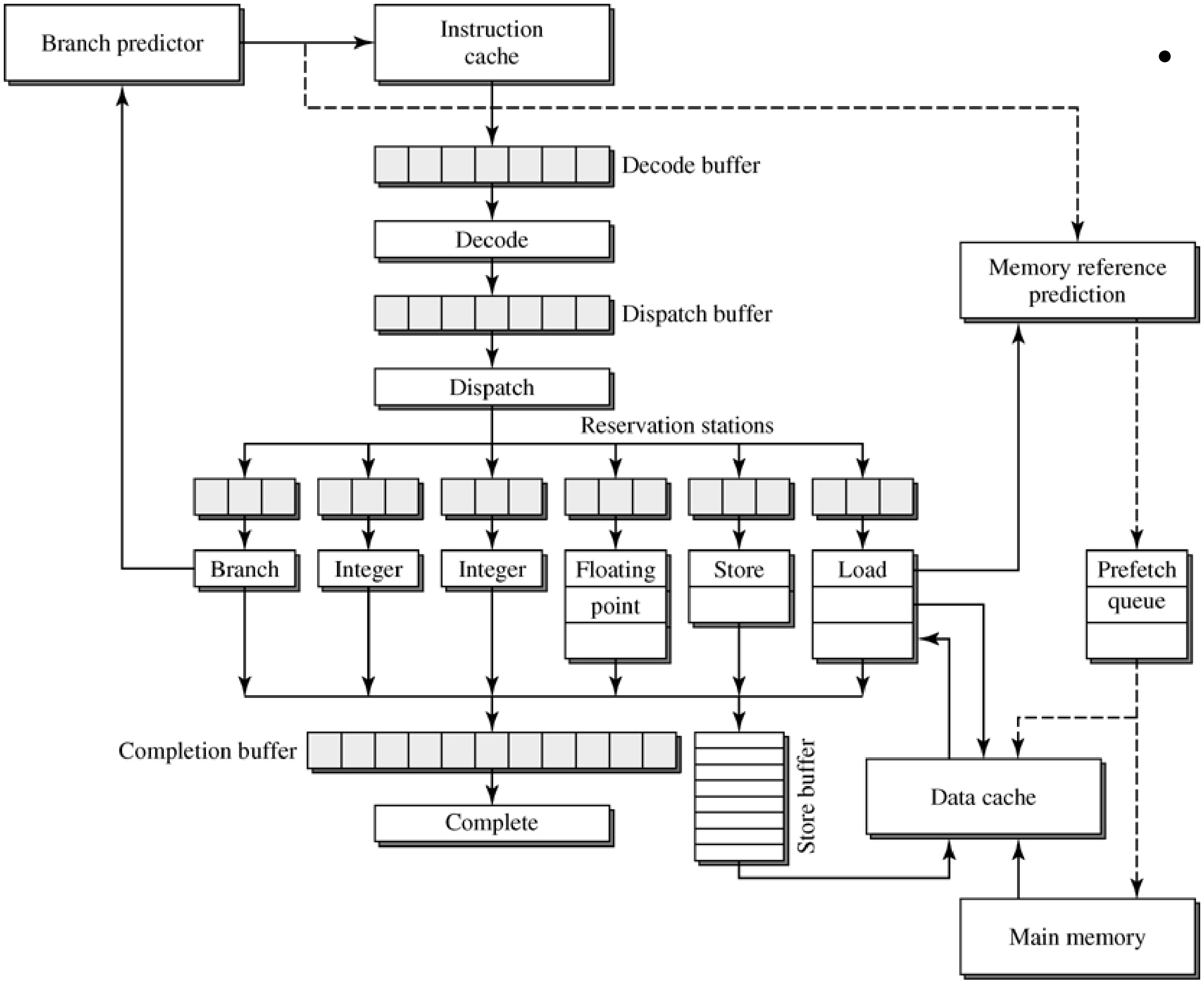
Prefetching cache / data (instruction) prefetching

- Terminology: …it depends only on the place where HW supporting prefetching is placed
- Idea is to fetch/fill cache lines before their content is addressed
- Memory access pattern differs for instructions and data (instruction cache vs. data cache)

Possible results of prefetching:

- useful prefetch (fetch done, hit follows)
- useless prefetch (data were in the cache, but replaced before hit or no hit follows)
- harmful prefetch (cache line has been replaced even that it is demanded again in the near future – cache pollution)

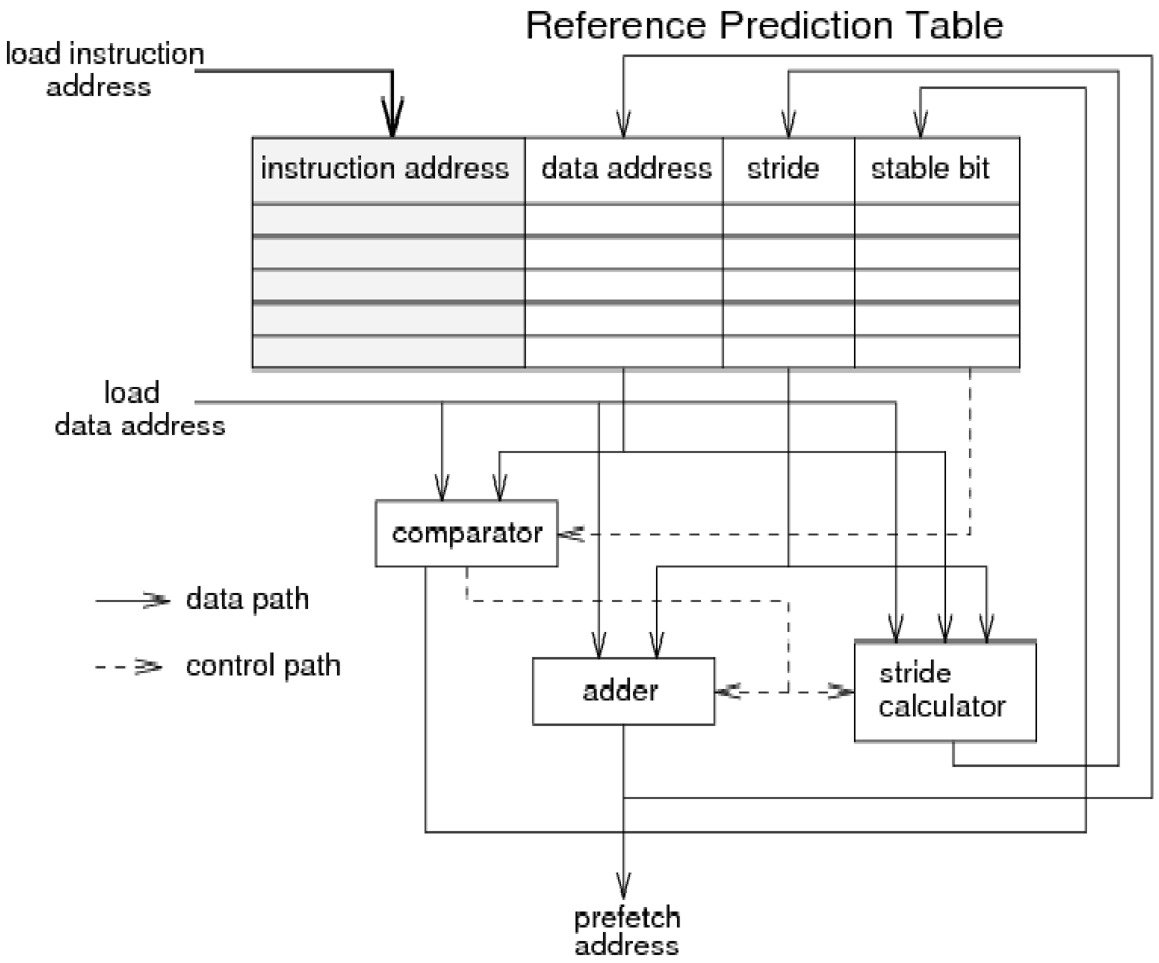
Prefetching cache / data (instruction) prefetching



- Load address prediction
- Load value prediction (previous lecture)

How to predict locations for prefetch? Example.

- Memory reference prediction



- HW techniques contributing to the maximization of instructions throughput have been presented until now – dynamic instructions scheduling and speculative execution

VLIW processors etc.

- Overview of static techniques contribution ILP grow follows, i.e., compiler assisted ILP

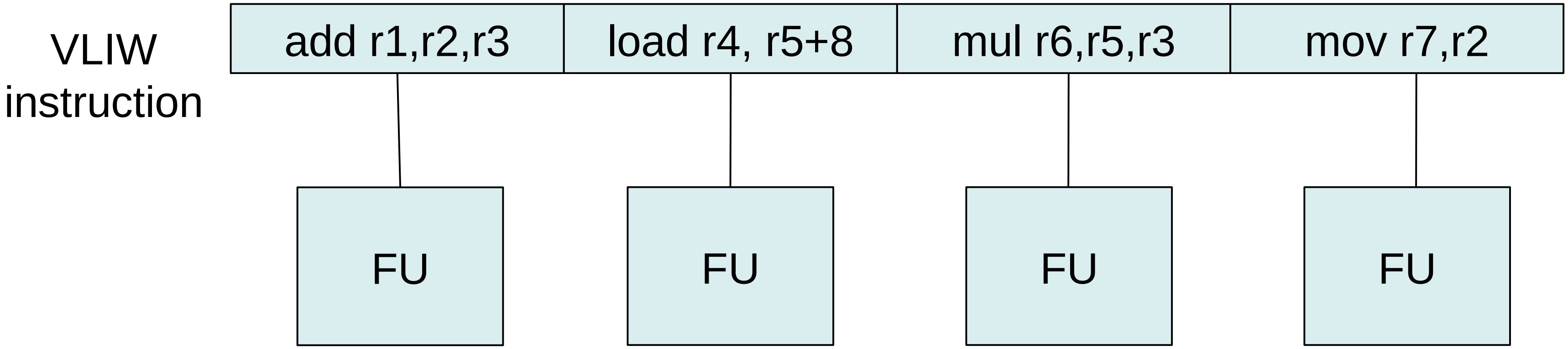
VLIW processors

- Very Long Instruction Word (4-16 instructions typically)
- VLIW architecture enables **parallel processing** (multiple original instructions/operations) by **single new instruction**.
- Instructions to be processed in parallel are scheduled in advance.
- In the compilation phase. (+ i -)
- VLIW is MIMD example in Flynn's classification
- Classic VLIW CPU does not include hazards detection – operations (sub-instructions) independence for single VLIW instruction is considered => simpler HW

VLIW and precise exception support

- Semantical unit for exception acceptance is single instruction (very long, or may it be better very wide) again.
- Fixed instructions format includes fields to encode multiple operations which are executed in parallel.

VLIW - Principle



- HW processed all operations encoded in instruction independently – fine-grain parallelism – parallelism on instructions level (ILP)
- The compiler is responsible for operations placement into instructions and extraction of ILP possibilities from the program

Example

Program for superscalar DLX vs. (V)LIW DLX

```
LF    F0,0(R1)
LF    F6,-4(R1)
LF    F10,-8(R1)
ADDF F4,F0,F2
LF    F14,-12(R1)
ADDF F8,F6,F2
LF    F18, -16(R1)
ADDF F12,F10,F2
SF   0(R1),F4
ADDF F16,F14,F2
SF   -4(R1),F8
ADDF F20,F18,F2
SF   - 8(R1),F12
SF   -12(R1),F16
SUBI   R1,R1,#20
BNEZ  R1,LOOP
SF   4(R1),F20
```

17 instructions
x 4B each
= 68B

```
LF    F0,0(R1)          NOP
LF    F6,-4(R1)         NOP
LF    F10,-8(R1)        ADDF F4,F0,F2
LF    F14,-12(R1)       ADDF F8,F6,F2
LF    F18, -16(R1)      ADDF F12,F10,F2
SF   0(R1),F4           ADDF F16,F14,F2
SF   -4(R1),F8          ADDF F20,F18,F2
SF   - 8(R1),F12        NOP
SF   -12(R1),F16        NOP
SUBI   R1,R1,#20        NOP
BNEZ  R1,LOOP           NOP
SF   4(R1),F20          NOP
```

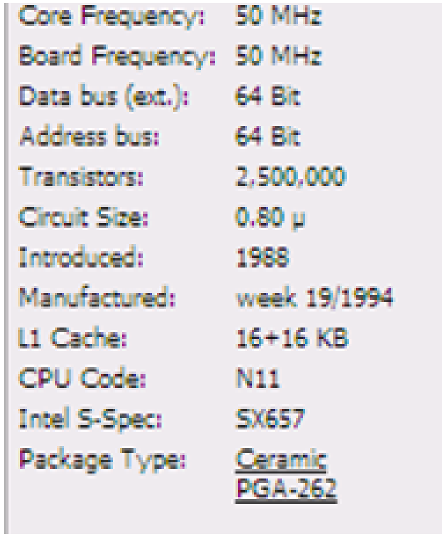
12 (long) instructions
x 8B each
= 96B

Source: Ing. Miloš Bečvář – Superpipelinové a Superskalární Procesory Procesory VLIW

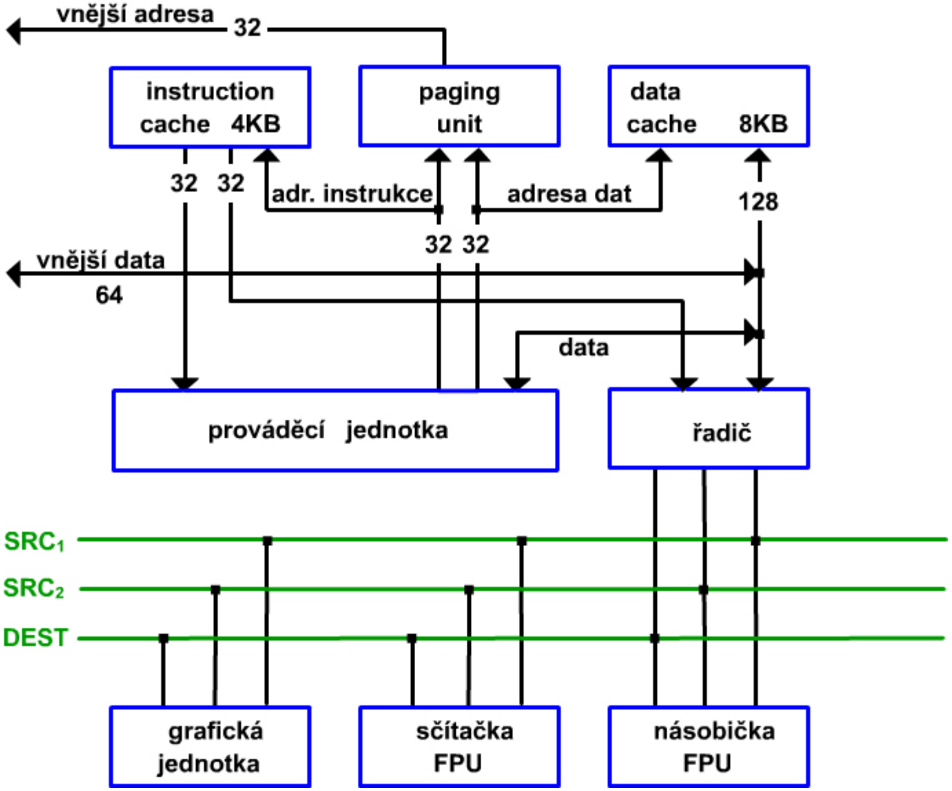
VLIW example

i860 XP
A80860XP-50
L4190197
SX657
(M)(C)1988 1990

94075906AC
MALAY
A 419



Core Frequency:	50 MHz
Board Frequency:	50 MHz
Data bus (ext.):	64 Bit
Address bus:	64 Bit
Transistors:	2,500,000
Circuit Size:	0.80 µ
Introduced:	1988
Manufactured:	week 19/1994
L1 Cache:	16+16 KB
CPU Code:	N11
Intel S-Spec:	SX657
Package Type:	Ceramic PGA-262



Another VLIW CPU example - TM3270 Media-Processor

- Designed for video and sound processing, 2005 year
- Variable instructions size from 2B to 28B

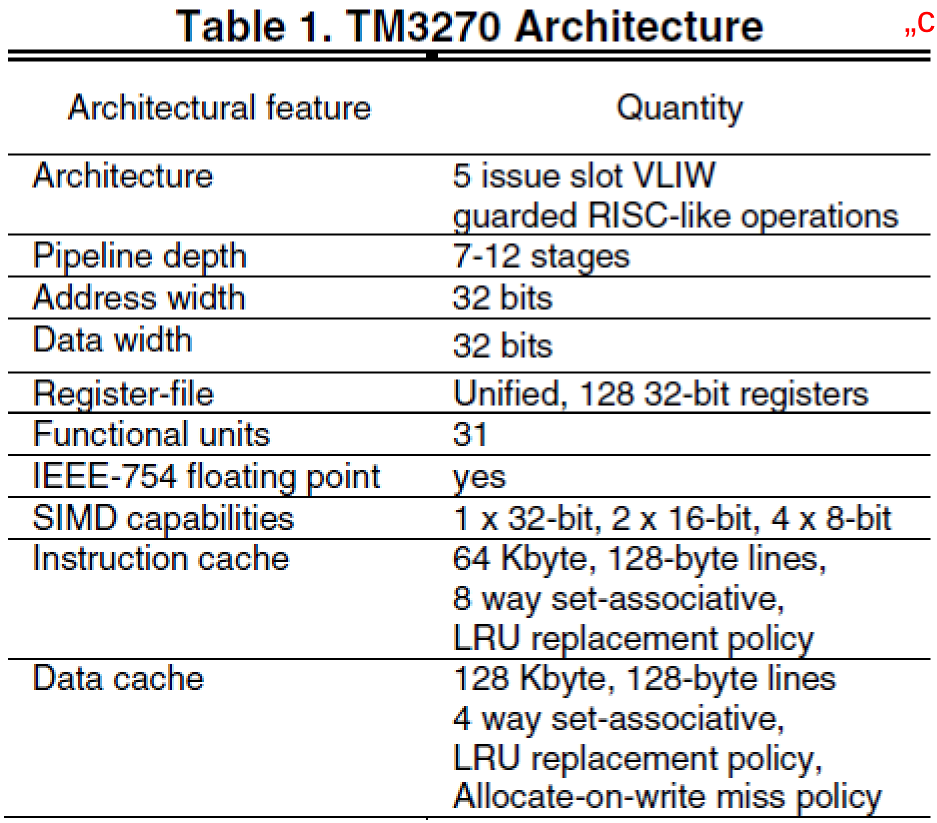
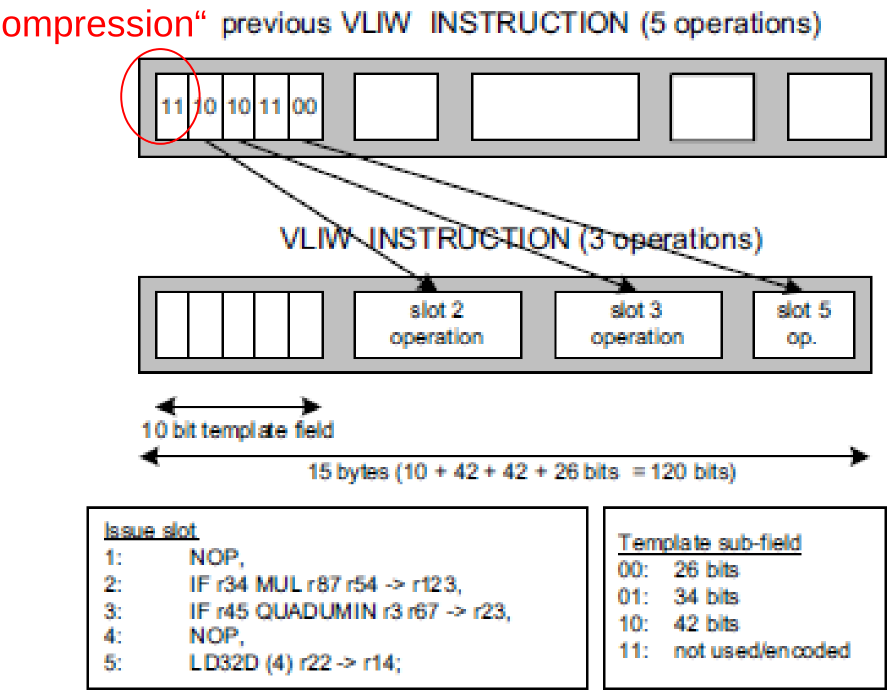


Table 1. TM3270 Architecture

Architectural feature	Quantity
Architecture	5 issue slot VLIW guarded RISC-like operations
Pipeline depth	7-12 stages
Address width	32 bits
Data width	32 bits
Register-file	Unified, 128 32-bit registers
Functional units	31
IEEE-754 floating point	yes
SIMD capabilities	1 x 32-bit, 2 x 16-bit, 4 x 8-bit
Instruction cache	64 Kbyte, 128-byte lines, 8 way set-associative, LRU replacement policy
Data cache	128 Kbyte, 128-byte lines 4 way set-associative, LRU replacement policy, Allocate-on-write miss policy


„compression“
previous VLIW INSTRUCTION (5 operations)
11
10
10
11
00
VLIW INSTRUCTION (3 operations)
slot 2 operation
slot 3 operation
slot 5 op.
10 bit template field
15 bytes (10 + 42 + 42 + 26 bits = 120 bits)

Issue slot
1: NOP,
2: IF r34 MUL r87 r54 -> r123,
3: IF r45 QUADUMIN r3 r67 -> r23,
4: NOP,
5: LD32D (4) r22 -> r14;

Template sub-field
00: 26 bits
01: 34 bits
10: 42 bits
11: not used/encoded

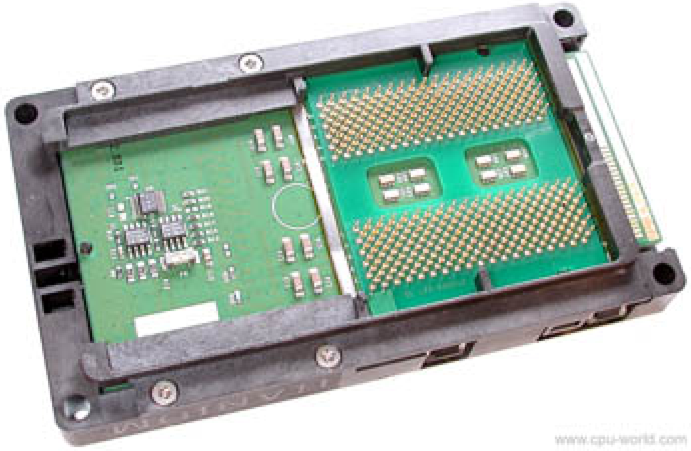
Figure 1. VLIW instruction encoding.

Modern DSP

- Why VLIW? Producer provides HW (processor) and corresponding SW support (compiler) …
- Superscalar processing
- Clock frequency more than 1 GHz
- Dual-level cache up to 8 MB
- SIMD
- VLIW – up to 8 instructions in the single cycle
- Special units designed for FFT and other signals processing

What is EPIC

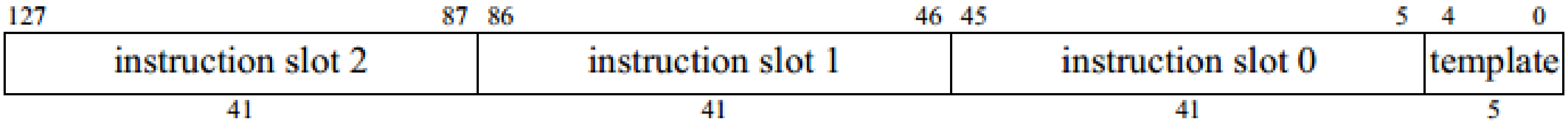
- Explicitly Parallel Instruction Computing
- Based and extends VLIW principles
- Representative is **Itanium** (original naming **IA-64**).
- It implements already described methods – speculation, branch prediction, and registers renaming.


www.cpu-world.com

EPIC and VLIW differences and what is new

- Bundle/packet – definition for the group of instructions which are packed to be executed in parallel
- The bundle can include „stop“ which indicates that there is a dependency between operations in the bundle
- SW prefetch (lfetch)
- Prediction – some mean of speculation
- Speculative load (ld.s, ld.sa,ld.c.nc, ld.c.clr,…),
- Move load to be initiated for execution earlier and later checking in the place of original load instruction location
- Issue the load earlier before the store and checking for aliasing later (the same address)

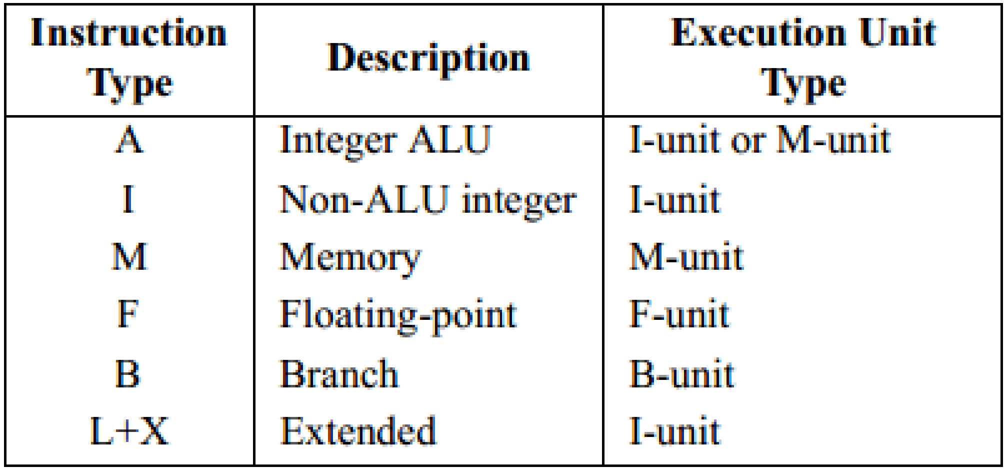
Intel Itanium – IA-64



Bundle Format

- IA-64 distinguishes 6 instruction types
- The bundle is composed from 3 instructions

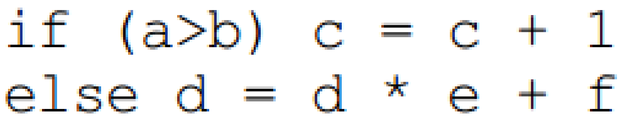
Relationship Between Instruction Type and Execution Unit Type



Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit

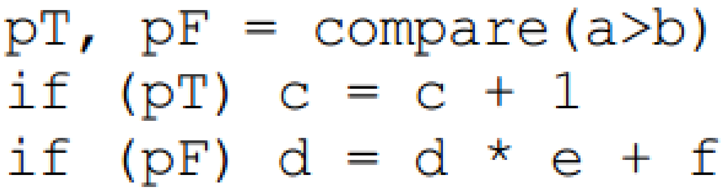
Prediction

- Original code:



```
if (a>b) c = c + 1
else d = d * e + f
```

- New code:



```
pT, pF = compare(a>b)
if (pT) c = c + 1
if (pF) d = d * e + f
```

- Predicate pT is set if the condition is true. pF predicate is complement to pT predicate
- The control dependency is converted to data dependency
- The another advantage is the possibility to pack instructions for parallel execution

Itanium – Explicitly Parallel Instruction Computing (EPIC)

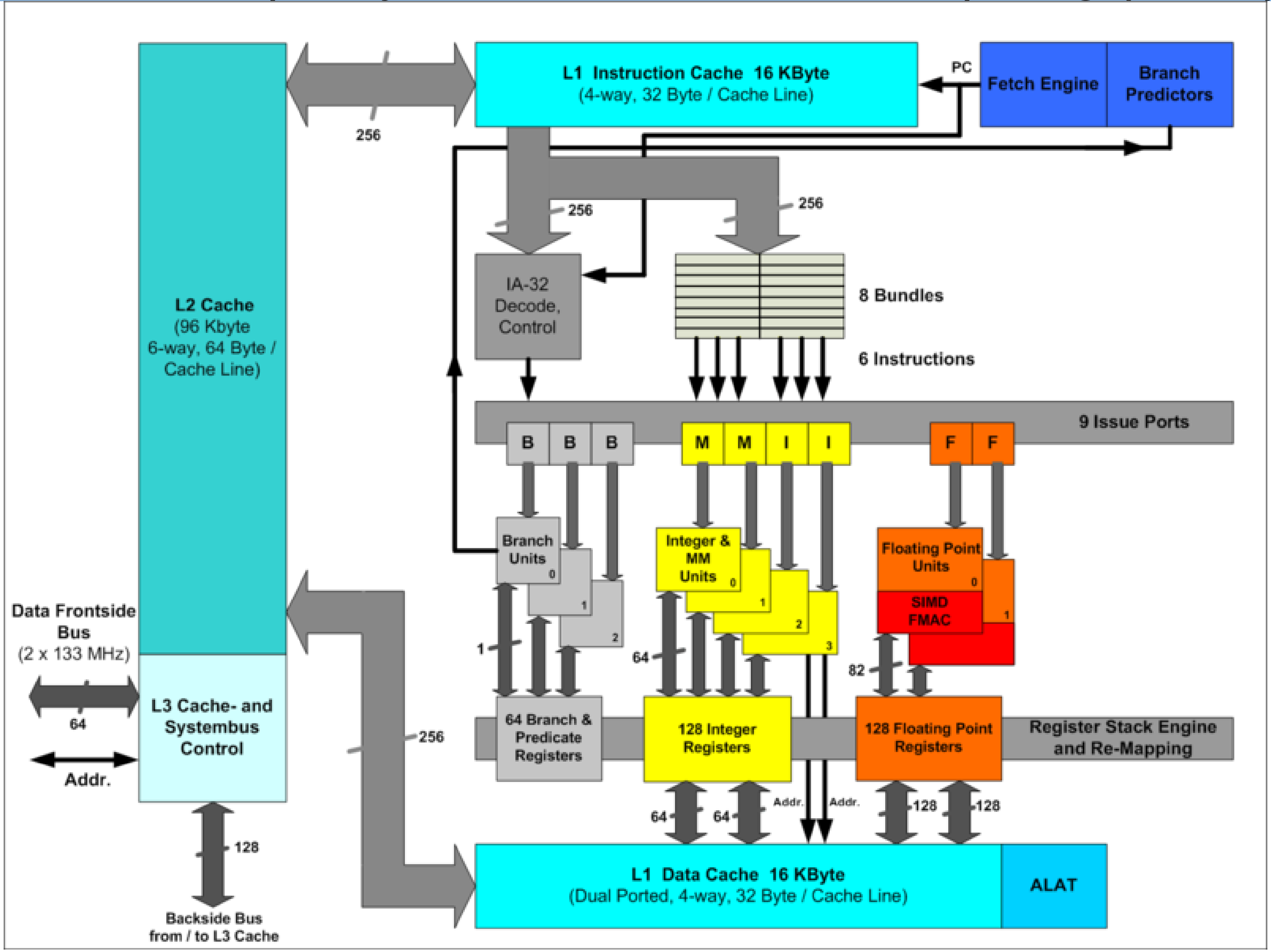


Figure source http://en.wikipedia.org/wiki/File:Itanium_arch.png

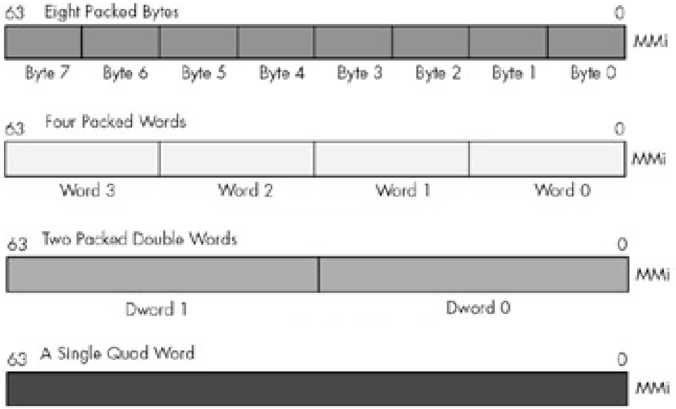
Itanium – IA-64 – EPIC

- It dramatically differs when compared to x86 (i x86-64)
- It is based on explicit ILP; parallelization is controlled by the compiler.
- It does not require additional HW for hazards detection and resolution – in theory, in practice it is required for deeper pipelines but is simpler and has more cycles to analyze inter-bundle dependencies.

Use of data parallelism

Profit and use of data parallelism, SIMD

- Single Instruction, Multiple Data, one of category from classic Flynn's taxonomy.
- One of broadly used SIMD implementation was the introduction of MMX (*MultiMedia eXtensions)* in x86.
- Initially for games and media
- Defines 8 64-bit registers MM0 … MM7
- MMX:
  - movd - MOVe Doubleword
  - paddb - adds an MMX register and another MMX register or memory as unsigned 8-bit bytes
  - paddw add an MMX register and another register or memory as unsigned 16-bit words
  - psubw subtracts unsigned 16-bit words



Vector instructions for x86 ISA

- This group includes:
- 3DNow! Od AMD
  - PAVGUSB – Packed 8-bit unsigned integer averaging
  - PFADD – Packed floating-point addition
  - PFMIN – Packed floating-point minimum
  - PREFETCH/PREFETCHW – Prefetch at least a 32-byte line into L1 data cache
- SSE (**Streaming SIMD Extensions**) and additional versions SSE2, SSE3, SSSE3, SSE4 from Intel – reaction to 3DNow introduction by AMD

SSE instruction - examples

- SSE3 instruction **ADDSUBPD** (*Add-Subtract-Packed-Double*)
  Input: { A0, A1 }, { B0, B1 }
  Output: { A0 − B0, A1 + B1 }
- SSE3 instruction **HADDPS** (*Horizontal-Add-Packed-Single*)
  Input: { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  Output: { A0 + A1, A2 + A3, B0 + B1, B2 + B3 }
- SSE4 Instruction **MPSADBW**:
  $|x_0-y_0|+|x_1-y_1|+|x_2-y_2|+|x_3-y_3|$, $|x_0-y_1|+|x_1-y_2|+|x_2-y_3|+|x_3-y_4|$,…, $|x_0-y_7|+|x_1-y_8|+|x_2-y_9|+|x_3-y_{10}|$
  8 times sum-absolute-difference (SAD), important for HighDefinition (HD) video codecs motion estimation
- SSE4 Instructions **DPPS**: Scalar products of Array of Structs

Vector instructions for non-x86 architectures

- PowerPC
  - POWER6, Power ISA v.2.03 – AltiVec
  - Power ISA v2.06 – VSX (Vector Scalar Extension)
  - Power ISA v3.0 – POWER9
- MIPS
  - MDMX (MaDMaX) and MIPS-3D
- Sparc
  - Visual Instruction Set VIS 1, VIS 2, VIS 2+, VIS 3 and VIS 4.

What next?

Loop

```
for (i=0; i<1000; i++)
    x[i] = x[i] + s;
```

Optimized to compare loop control variable to zero as

```
i = 999;
do x[i] = x[i] + s; while(i--);
```

```
Loop:   LD      F0,0(R1)      ; F0 = vector element (x base skipped)
        NOP
        ADDD    F4,F0,F2      ; add scalar from F2
        SD      0(R1),F4      ; store result
        SUBI    R1,R1,8       ; decrement pointer 8bytes (DW)
        BNEZ    R1,Loop       ; branch R1!=zero
        NOP                   ; delayed branch slot
```

Loop, 7 cycles

Loop unrolling

```
1 Loop:  LD      F0,0(R1)
2        LD      F6,-8(R1)
3        LD      F10,-16(R1)
4        LD      F14,-24(R1)
5        ADDD    F4,F0,F2
6        ADDD    F8,F6,F2
7        ADDD    F12,F10,F2
8        ADDD    F16,F14,F2
9        SD      0(R1),F4
10       SD      -8(R1),F8
11       SD      -16(R1),F12
12       SUBI    R1,R1,32
13       BNEZ    R1,LOOP
14       SD      8(R1),F16          ; 8-32 = -24
```

- NOP instructions removed
- Suitable even for CPU which does not support registers renaming (Registers renamed by the compiler.)
- Stall cycles minimized even for scalar processor

Loop unrolled 4x, 14 cycles

Execution on superscalar processor

- Imagine the Tomasulo algorithm in action

Iteration	*Instructions*	*Issues*	*Executes*	*Writes result*
no.			***clock-cycle number***	
1	**LD F0,0(R1)**	**1**	**2**	**4**
1	**ADDD F4,F0,F2**	**1**	**5**	**8**
1	**SD 0(R1),F4**	**2**	**9**	
1	**SUBI R1,R1,#8**	**3**	**4**	**5**
1	**BNEZ R1,LOOP**	**4**	**5**	
2	**LD F0,0(R1)**	**5**	**6**	**8**
2	**ADDD F4,F0,F2**	**5**	**9**	**12**
2	**SD 0(R1),F4**	**6**	**13**	
2	**SUBI R1,R1,#8**	**7**	**8**	**9**
2	**BNEZ R1,LOOP**	**8**	**9**	

4 cycles for iteration, NOP?

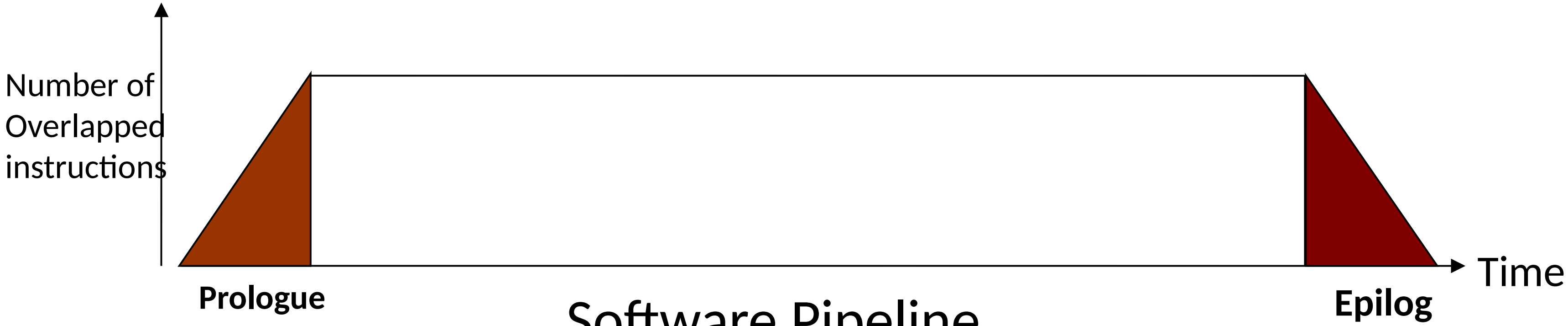
Loop unrolling for VLIW

Memory reference 1	Memory reference 2	FP operation 1	FP op. 2	Int. op/ branch	Clock
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
ADDD F20,F18,F2		ADDD F24,F22,F2			5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

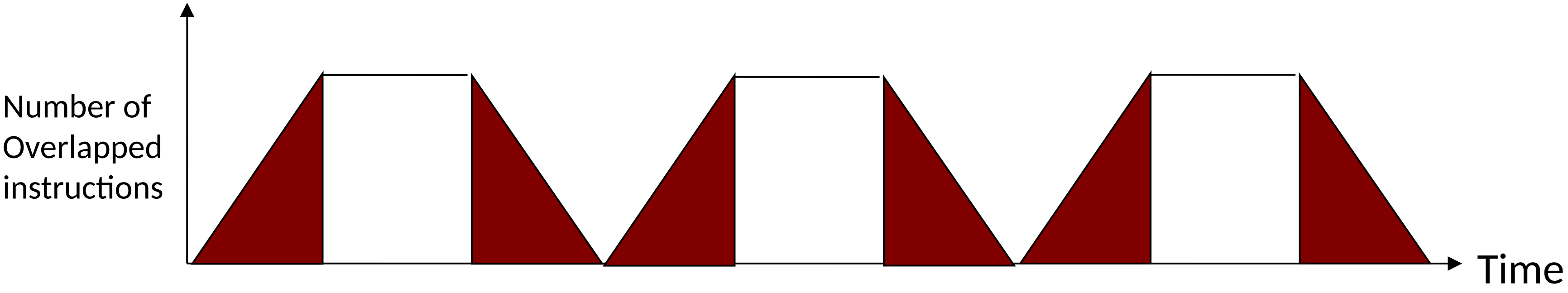
1 row – 1 instruction

Loop unrolled 7x, 9 cycles

Software Pipeline



Software Pipeline



Loop unrolled

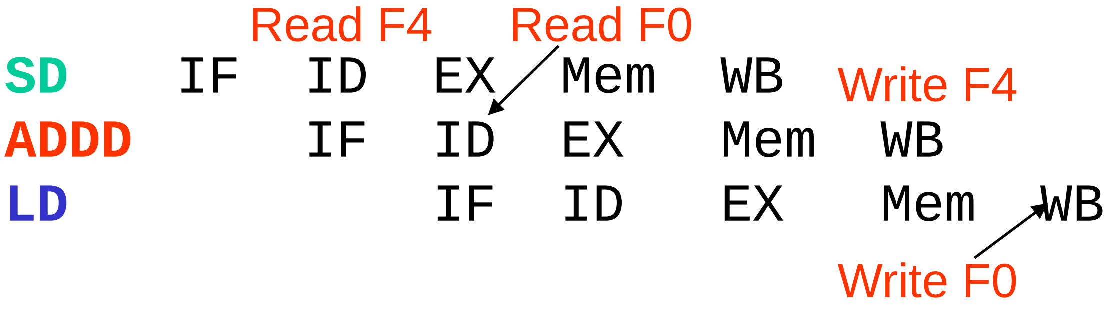
SW Pipelining - Example

Before: Unrolled 3 times

```
1   LD  F0,0(R1)
2   ADDD   F4,F0,F2
3   SD  0(R1),F4
4   LD  F6,-8(R1)
5   ADDD   F8,F6,F2
6   SD  -8(R1),F8
7   LD  F10,-16(R1)
8   ADDD   F12,F10,F2
9   SD  -16(R1),F12
10  SUBI   R1,R1,#24
11  BNEZ   R1,LOOP
```

After: Software Pipelined

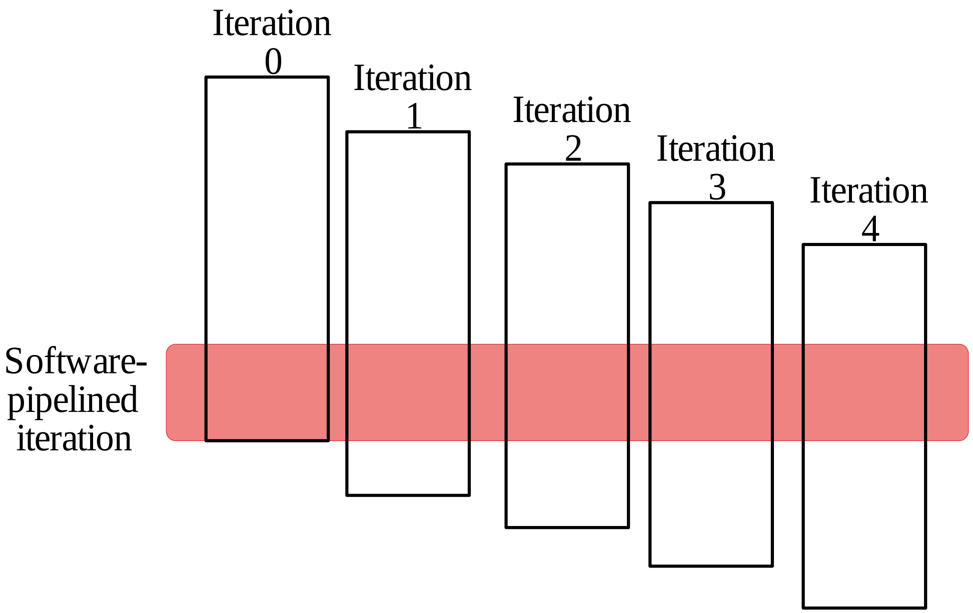
```
    LD  F0,0(R1)
    ADDD   F4,F0,F2
    LD  F0,-8(R1)
1   SD  0(R1),F4;     Stores M[i]
2   ADDD   F4,F0,F2;  Adds to M[i-1]
3   LD  F0,-16(R1);   loads M[i-2]
4   SUBI   R1,R1,#8
5   BNEZ   R1,LOOP
    SD  0(R1),F4
    ADDD   F4,F0,F2
    SD  -8(R1),F4
```



```
              Read F4      Read F0
SD      IF    ID     EX     Mem    WB     Write F4
ADDD          IF     ID     EX     Mem    WB
LD                   IF     ID     EX     Mem    WB
                                          Write F0
```

SW Pipelining – Symbolic Loop Unrolling

- If individual **cycle iterations are independent** (instructions within the cycle may be dependent), we can achieve an increase in ILP by grouping instructions from different iterations.
- SW pipelining reorganizes the loops (≈ the Tomasulo algorithm over the expanded loop)
- We will achieve the greatest middle degree of parallelization only with a small increase in code



Example how to eliminate dependency between iterations

- Dependency between iterations is caused by B

OLD:

```
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + B[i];    /* S1 */
    B[i+1] = C[i] + D[i];} /* S2 */
```

NEW:

```
A[1] = A[1] + B[1];
for  (i=1; i<=99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] =  + A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Used sources:

1. **Shen, J.P., Lipasti, M.H.: Modern Processor Design: Fundamentals of Superscalar Processors, First Edition, New York, McGraw-Hill Inc., 2005**
2. Sunil Kim, Alexander V. Veidenbaum: Second level cache organization for data prefetching. 1996
3. David A. Patterson: Lecture 5: VLIW, Software Pipelining, and Limits to ILP. Computer Science 252, Fall 1996.
4. Ioannis Papaefstathiou: Advanced Computer Architecture - Chapter 4. Advanced Pipelining. CS 590.25 Easter 2003.
5. van de Waerdt, J.-W.; Vassiliadis, S.; et al. "The TM3270 media-processor,"*Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on* , vol., no., pp.12 pp.,342, 16-16 Nov. 2005
6. http://www.csee.umbc.edu/portal/help/architecture/aig.pdf
7. http://www.cs.cmu.edu/afs/cs/academic/class/15740-f03/public/doc/discussions/uniprocessors/ia64/mpr_ia64_isa_may99.pdf