# VI. Implicit Parallelism - Instruction Level Parallelism. Pipeline Superscalar & Vector Processors.

March 16th, 2009

## Content

- Pipelining
- Vector Processors
- Superscalar Processors
- Programming issues

## Traditional serial scalar von Neumann architecture

- Provides single control flow with serially executed instructions operating on scalar operands.
- The processor has one instruction execution unit (IEU).
- Execution of an instruction can be only started after execution of the previous instruction in the flow is terminated.
- Except for a relatively small no. of special instructions for data transfer between main memory and registers, the instructions take operands from and put results to scalar registers
  - a scalar register is a register that holds a single integer or float no.
- The total time of program execution is equal to the sum of execution times of the instructions.
- Performance of that architecture is determined by the clock rate.
- All three components – processor, memory, and datapath – present bottlenecks to the overall processing rate of a computer system.
  - Architectural innovations over the years have addressed these bottlenecks.
  - One of the most important innovations is multiplicity – in processing units, datapaths, and memory units.

## Pipelining

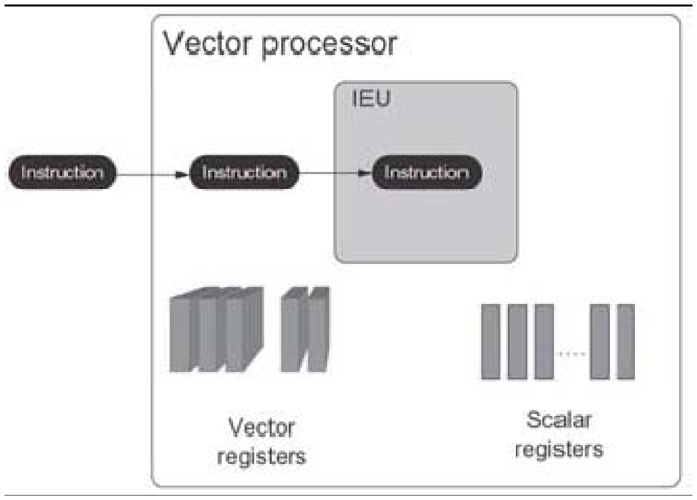
- By overlapping various stages in instruction execution (fetch, schedule, decode, operand fetch, execute, store etc), enables faster execution.
- The assembly-line analogy works well for understanding pipelines.
  - If the assembly of a car taking 100 time units can be broken into 10 pipelined stages of 10 units each, the assembly line can produce a car every 10 time units!
    - Represents a 10-fold speedup over producing cars entirely serially, one after the other.
  - It is also evident from this example that to increase the speed of a single pipeline, one would break down the tasks into smaller and smaller units, thus lengthening the pipeline and increasing overlap in execution.
  - In the context of processors, this enables faster clock rates since tasks are smaller.
- Example: Pentium 4 operates at 2.0 GHz and has a 20 stage pipeline.
- On modern machines, essentially all operations are pipelined:
  - several hardware stages are needed to do any computation.
  - It is possible to do multiple operands concurrently: as soon as a low order digit for one operand pair is computed by one stage, that stage can be used to process the same low order digit of the next operand pair.
- This notion of pipelining operations was also not invented yesterday
  - the University of Manchester *Atlas* project implemented such arithmetic pipelines as early as 1962.

## Pipelining

- The terminology is an analogy to a short length of pipe into which one starts pushing marbles:
    - Imagine that the pipe will hold $L$ marbles, which will be symbolic for stages necessary to process each operand.
    - To do one complete operation on one operand pair takes $L$ steps.
    - With multiple operands, we can keep pushing operands into the pipe until it is full, after which one result (marble) pops out the other end at a rate of one result/cycle.
    - By this device, instead $n$ operands taking $L$ cycles each, that is, a total of $n \cdot L$ cycles, only $L + n$ cycles are required as long as the last operands can be flushed from the pipeline once they have started into it.
    - The resulting speedup is $n \cdot L/(n + L)$, that is, $L$ for large $n$.
- Speed of a pipeline is limited by the largest atomic task in the pipeline.
- In typical instruction traces, every fifth to sixth instruction is a branch instr.
    - Long instruction pipelines therefore need effective techniques for predicting branch destinations so that pipelines can be speculatively filled.
    - The penalty of a misprediction increases as the pipelines become deeper since a larger number of instructions need to be flushed.
    - These factors place limitations on the depth of a processor pipeline and the resulting performance gains.

# Vector Processor

- Provides a single control flow with serially executed instructions operating on both vector and scalar operands.
- The parallelism of this architecture is at the instruction level.
- Like the serial scalar processor:
  - it has only one IEU: does not begin executing the next instruction until the execution of the current one is completed.
- Unlike the serial scalar processor:
  - its instructions can operate both on scalar operands and on vector operands.
- The vector operand:
  - is an ordered set of scalars that is generally located on a vector register.

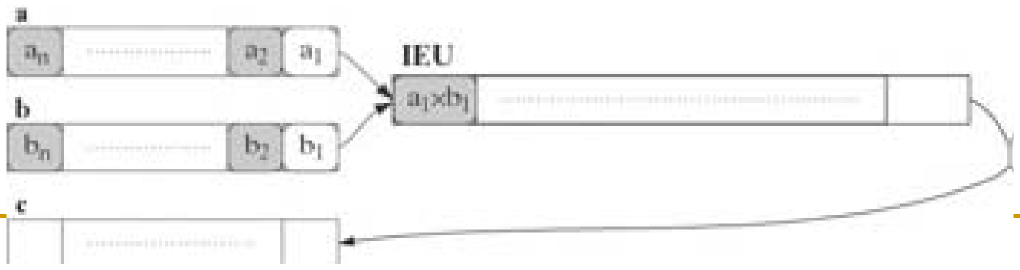


-implementations of this arch.: ILLIAC-IV, STAR-100, Cyber-205, Fujitsu VP 200, ATC,
-most elegant is the vector compute
Cray-1 in 1976.
-employs a data pipeline to execute the vector instrucs.

# Example – vector processor (1)

- Multiplication of two vector operands,
  - Element-wise extensions of the corresponding scalar operation.
- Operands from vector registers **a** and **b** and putting the result on vector register **c**.
- The instruction is executed by a pipelined unit able to multiply scalars.
- The multiplication of two scalars is partitioned into *m* stages, and the unit can simultaneously perform different stages for different pairs of scalar elements of the vector operands.
- The execution of vector instruction **c = a x b**, where **a**, **b**, and **c** are *n*-element vector registers, by the pipelined unit can be summarized as follows:

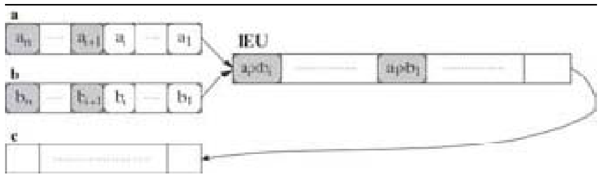
1. first step, the unit performs stage 1 of the multiplication of elements *a*1 and *b*1:



## Example – vector processor (2)

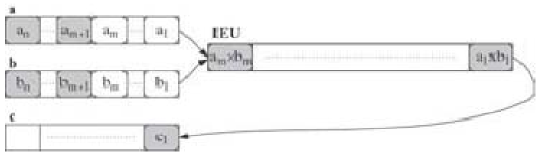
*i*th step (*i* = 2,…, *m* - 1), the unit performs in parallel:

- stage 1 of the multiplication of elements *ai* and *bi*,
- stage 2 of the multiplication of elements *ai-1* and *bi-1*, …
- stage *i* of the multiplication of elements *a*1 and *b*1,

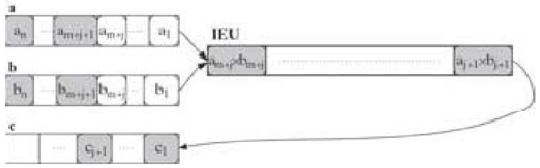


Following steps:

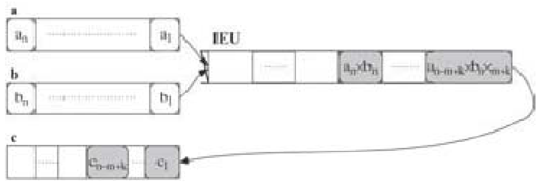
*m*th



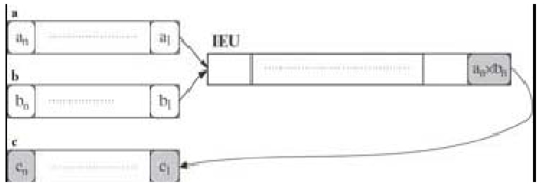
*(m+j)*th



*(n+k-1)*th



*(n+m-1)*th
(Final)



## Example – vector processor (3)

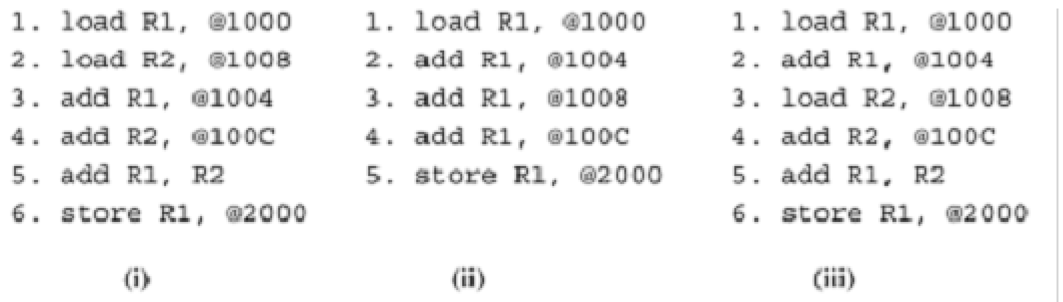
- In total, it takes $n + m - 1$ steps to execute this instruction.
- The pipeline of the unit is fully loaded only from the *m*th to the *n*th step of the execution.
- Serial execution of $n$ scalar multiplications with the same unit would take $n \times m$ steps.
- The speedup provided by vector instruction is $S = n \times m / (n+m-1)$
- If $n$ is large enough, the speedup is approximately equal to the length of the unit's pipeline, $S \approx m$.
- Vector architectures are able to speed up such applications, a significant part of whose computations falls into the basic element-wise operations on arrays.
- Vector architecture includes the serial scalar architecture as a particular case ($n = 1$, $m = 1$).

## Superscalar execution

- An obvious way to improve instruction execution rate beyond this level is to use multiple pipelines.
  - During each clock cycle, multiple instructions are piped into the processor in parallel.
  - These instructions are executed on multiple functional units.
- The ability of a processor to issue multiple instructions in the same cycle is referred to as superscalar execution
- Example:
  - Consider a processor with two pipelines and the ability to simultaneously issue two instructions.
  - These processors are sometimes also referred to as **super-pipelined processors**.
  - Two issues per clock cycle => it is also referred to as two-way superscalar or dual issue execution.

## Example

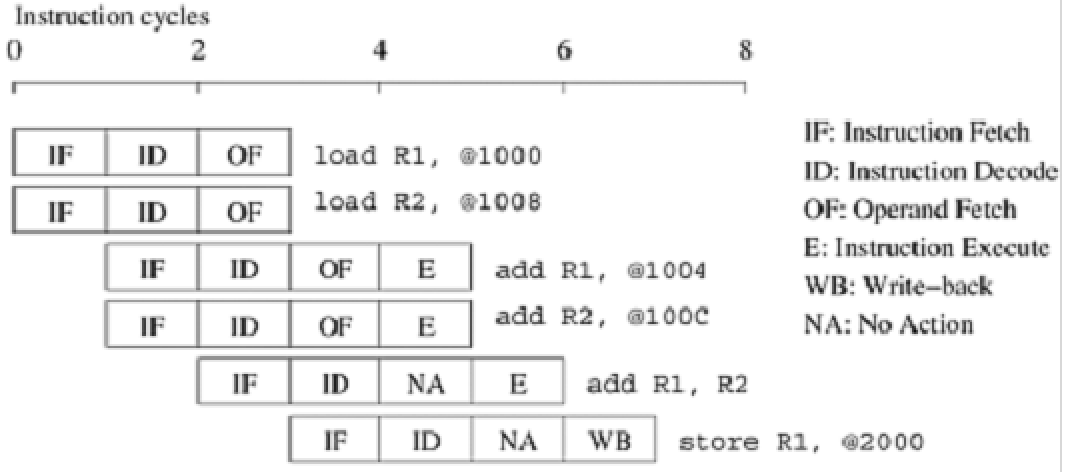
- adding four numbers.
- The first and second instructions are independent and are issued concurrently:
  - load R1, @1000
  - load R2, @1008 at t=0
- The schedule assumes that each memory access takes a single cycle. In reality, this may not be the case.



```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000

      (i)                    (ii)                   (iii)
```

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.

## Superscalar issue: dependency between instructions

- instructions in a program may be related to each other.
- Data dependency:
  - The results of an instruction may be required for subsequent instructions.
  - The dependencies must be resolved before simultaneous issue of instrs.
  - ⇒ since the resolution is done at runtime, it must be supported in hardware.
  - ⇒ the amount of instruction level parallelism in a program is often limited and is a function of coding technique.
  - in many cases it is possible to extract more parallelism by reordering the instructions and by altering the code
- Resource dependency:
  - finite resources shared by various pipelines.
  - Example:
    - ❑ co-scheduling of two floating point operations on a dual issue machine with a single floating point unit.
    - ❑ Instrs cannot be scheduled together since both need the floating point unit.
- Branch dependency or procedural dependency:
  - Since the branch destination is known only at the point of execution, scheduling instructions a priori across branches may lead to errors.

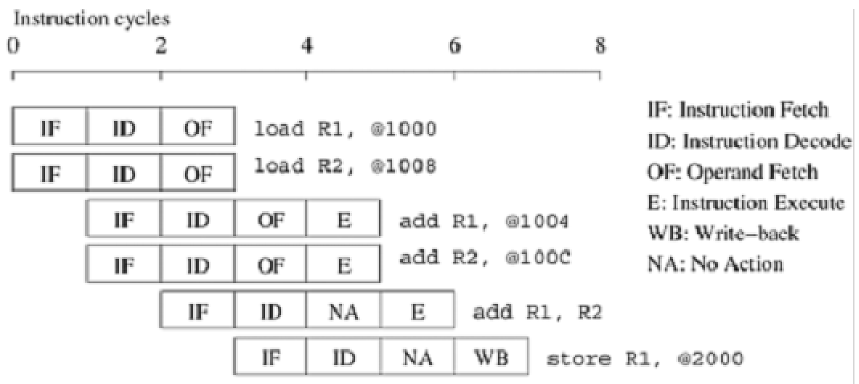
# Microprocessors capabilities

- The ability of a processor to detect and schedule concurrent instructions is critical to superscalar performance.
- From the example:
  - detect that it is possible to schedule the third instruction – load R2, @1008 – with the first instruction.
- Most current microprocessors are capable of out-of order issue and completion
  - Model, referred as ***dynamic instruction issue***, exploits maximum instr.level parallelism
  - Procs.uses a window of instrs from which it selects instructions for simultaneous issue
    - This window corresponds to the look-ahead of the scheduler.
- Current microprocs typically support up to four-issue superscalar execution
- Due to:
  - limited parallelism,
  - resource dependencies, or
  - the inability of a processor to extract parallelism,

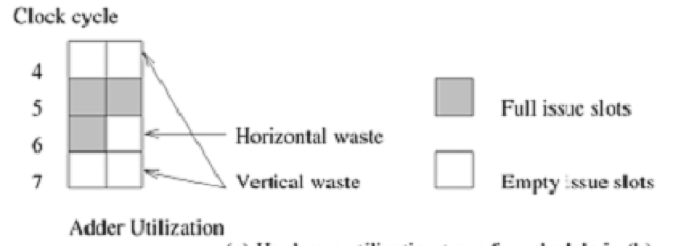
  the resources of superscalar processors are heavily under-utilized.
- Parallelism extracted by superscalar procs is limited by instruction look-ahead.
  - The hardware logic for dynamic dependency analysis is typically in the range of 5-10% of the total logic on conventional microprocessors
    - about 5% on the four-way superscalar Sun UltraSPARC.
  - Complexity grows roughly quadratically with the no.issues & can become a bottleneck.

# Superscalar arch. performance limited by the available instruction level parallelism

- In (c) 2 execution units (multiply-add units),
- Illustrates several zero-issue cycles
  - cycles in which the floating point unit is idle).
  - These are essentially wasted cycles from the point of view of the execution unit.
- If, during a particular cycle, no instructions are issued on the execution units, it is referred to as vertical waste
- If only part of the execution units are used during a cycle, it is termed horizontal waste.
- In example, we have two cycles of vertical waste and one cycle with horizontal waste.
  - In all, only 3 of the 8 available cycles are used for computation.
  - ⇒ code fragment will yield no more than three-eighths of the peak rated FLOP count of the processor.



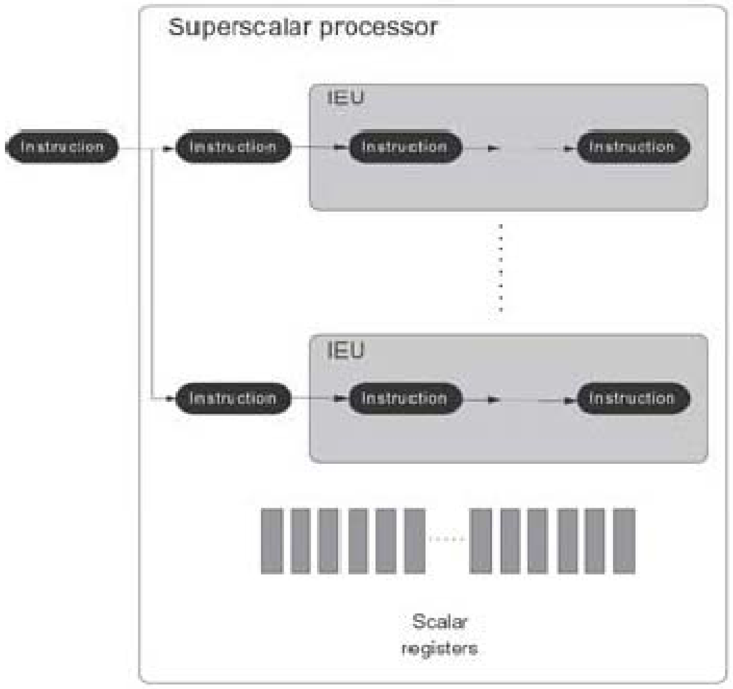
(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

## Superscalar Processor and scalar operands

- The superscalar processor provides a single control flow with instructions operating on scalar operands and being executed in parallel
- The superscalar processor has several instruction execution units executing instructions in parallel.
- Except for a small number of special instructions for data transfer between main memory and registers, the instructions operate on scalar operands located on the scalar registers.
- CDC 6600 (1964) was the first processor with several IEUs functioning in parallel.
- CDC 7600 (1969) was the first processor with several pipelined IEUs functioning in parallel.
- Nowadays practically all manufactured microprocessors have many IEUs, each of which is normally a pipelined unit, and we can hardly imagine other microprocessor architectures.



Two successive instructions can be executed in parallel by two different IEUs if they do not have conflicting operands, that is,
• if they do not write in the same register and
• neither instruction uses a register in which the other writes.

## Superscalar and pipelines

- A dispatcher unit directing instructions to relevant IEUs based on their type.
- Each IEU is characterized by the set of instructions that it executes.
- The entire set of instructions is divided into a number of nonintersecting subsets each associated with some IEU.
- Additionally each IEU can be a pipelined unit:
  - it can simultaneously execute several successive instructions, each being on its stage of execution.
- Example:
  - a pipelined unit executing a no. of successive independent instrucs.
  - Let the pipeline of the unit consist of *m* stages.
  - Let *n* successive instrucs *I*1,…, *In*, be performed by the unit.
  - Instruction *Ik* takes operands from registers *ak*, *bk* and puts the result on register *ck* (*k* = 1,…, *n*).
  - Let no two instructions have conflicting operands.

# Example

- At the first step, performs stage 1 of instruction *I*1:
- At the *i*th step (*i* = 2,…, *m* - 1), the unit performs in parallel stage 1 of instruction *Ii*, stage 2 of instruction *Ii-1*, etc., and stage *i* of instruction *I*1,

...

- At the (*n* + *m* - 1)-th step, the unit only performs the final stage *m* of instruction *In; after completion* of this, register *cn* contains the result of instruction *In*

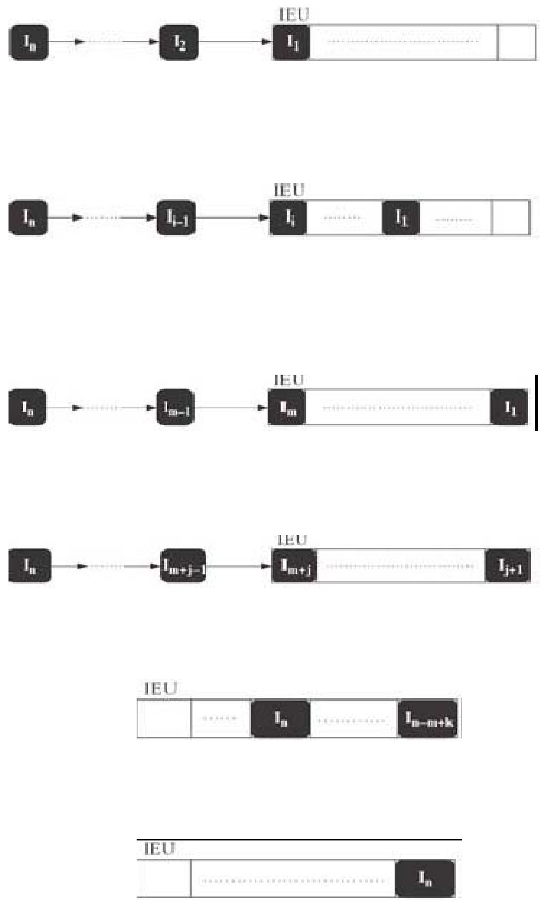
It takes (*n* + *m*-1) steps to execute *n* instruction.

The pipeline of the unit is fully loaded only from the *m*th to the *n*th step of the execution.

Strictly serial execution by the unit of *n* successive instructions takes *n x m* steps.

The maximal speedup provided by this pipelined unit is *S*= *n* x *m / (n*+*m*-1*).*

If *n* is large enough, the speedup *S*IEU ≈ *m*.



# Very Long Instruction Word Processors

- An alternate concept for exploiting instruction-level parallelism
- Relies on the compiler to resolve dependencies &resource availability at compile time
    - Instructions that can be executed concurrently
        - are packed into groups and
        - parceled off to the processor as a single long instruction word

      to be executed on multiple functional units at the same time.
    - Additional parallel instructions are made available to the compiler to control parallel execution
- First used in Multiflow Trace (1984) & subsequently as a variant in the Intel IA64 arch.
- Advantages:
    - Scheduling is done in software
    - The decoding and instruction issue mechanisms are simple
    - The compiler
        - has a larger context from which to select instructions and
        - can use a variety of transformations to optimize parallelism when compared to a hardware issue unit.
- Disadvantages:
    - compilers do not have the dynamic program state (e.g., the branch history buffer) available to make scheduling decisions.
        - reduces the accuracy of branch and memory prediction
    - Other runtime situations such as stalls on data fetch because of cache misses are extremely difficult to predict accurately.
        - limits the scope and performance of static compiler-based scheduling.
    - Performance is sensitive to the compilers' ability to detect data and resource dependencies and read and write hazards, and to schedule instructions for maximum parallelism.
- Limited to smaller scales of concurrency in the range of four- to eight-way parallelism.

## Common view: Vector & Superscalar procs

- Vector and superscalar archs. are united in a group for a no. reasons:

1. Successful vector archs are close to superscalar archs both ideologically and in implementation.
   - a vector pipelined unit is a specialized clone of the general-purpose superscalar pipelined unit, which is optimized for pipelined execution of *n* successive instructions performed as the same operation but on different operands.
     - optimization: the vector pipelined unit does not need a decoding stage and tuses a more effective data pipeline instead of the instruction pipeline.
2. The design of some advanced superscalar processors, such as Intel i860, is obviously influenced by the vector-pipelined architecture.
3. Share the same programming model
   - A good portable progr. that can take full advantage of the performance potential of superscalar procs is defined in the same way as for vector procs.
     - A good program for vector processors? It is the program's use of a wide range of vector instructions that implement basic operations on arrays.
     - A program intensively using basic ops on arrays is perfectly suitable for superscalar processors in that it allows a very-high level of utilization of their pipelined units.

- Unlike vector procs, superscalar procs allow more sophisticated mixtures of ops to efficiently load their pipelined units than just basic array ops
  - even if the superscalar arch looks richer than the vector arch, the real programming model used for superscalar procs should be the same as for vector procs.

## Loop unrolling – an programming example

The simple loop

```
    for(i=0;i<n;i++){ b[i] = f(a[i]);}
```

is expanded into segments of length (say) *m i*'s.

```
    for(i=0;i<n;i+=m){
            b[i ] = f(a[i ]);
            b[i+1] = f(a[i+1]);
            …
            b[i+m-1] = f(a[i+m-1]);
            }
    /* residual segment res = n mod m */
    nms = n/m; res = n%m;
    for(i=nms*m;i<nms*m+res;i++){
        b[i] = f(a[i]);
    }
```

The first loop processes nms segments, each of which does m operations f(a[i]).

Our last loop cleans up the remaining (a residual segment).

Sometimes this residual segment is processed first, sometimes last (as shown) or for data alignment reasons, part of the res first, the rest last.

We will refer to the instructions which process each f(a[i]) as a **template**.

Optimization: choose an appropriate depth of unrolling *m* which permits squeezing all the *m* templates together into the tightest time grouping possible.

## Pre-fetching as stage of loop unrolling

- Prefetching data within the segment which will be used by subsequent segment elements in order to hide memory latencies.
  - to hide the time it takes to get the data from memory into registers.
  - Such data pre-fetching was called bottom loading in former times.
- Pre-fetching in its simplest form is for *m* = 1 and takes the form

```
t = a[0]; /* prefetch a[0] */
for(i=0;i<n-1; ){
        b[i] = f(t);
        t = a[++i]; /* prefetch a[i+1] */
}
b[n-1] = f(t);
```

  where one tries to hide the next load of a[i] under the loop overhead.

- Loop unrolling purpose: hide latencies, in particular, the delay in reading data from memory
  - Unless the stored results will be needed in subsequent iterations of the loop (a data dependency), these stores may always be hidden: their meanderings into memory can go at least until all the loop iterations are finished.

# Instruction Scheduling with Loop Unrolling

- Consider the following operation to be performed on an array **A**: **B** = *f*(**A**), where *f*(·) is in two steps: *Bi* = *f*2(*f*1(*Ai*)).
- Each step of the calculation takes some time and there will be **latencies** in between them where results are not yet available.
- If we try to perform multiple *i*s together, *Bi* = *f*(*Ai*) and *Bi*+1 = *f*(*Ai*+1),
  - the various operations: memory fetch, *f*1 and *f*2, might run concurrently,
  - we could set up two **templates** and try to align them.
    - By starting the *f*(*Ai*+1) operation steps one cycle after the *f*(*Ai*), the two templates can be merged together.
- Each calculation *f*(*Ai*) and *f*(*Ai*+1) takes some number (say *m*) of cycles.
  - If these two calculations ran sequentially, they would take twice what each one requires, that is, $2 \cdot m$.
  - By merging the two together and aligning them to fill in the gaps, they can be computed in $m + 1$ cycles.
- This will work only if:
  1. the separate operations can run independently and concurrently,
  2. if it is possible to align the templates to fill in some of the gaps, and
  3. there are enough registers.
     - If there are only eight registers, alignment of two templates is all that seems possible at compile time.

# Data dependency

- Look at the following loop in which *f*(*x*) is some arbitrary function,

  ```
  for(i=m;i<n;i++){ x[i]=f(x[i-k]); }
  ```

  x[i-k] must be computed before x[i] when *k* > 0

- Counterexample:

  ```
  for(i=m;i<n;i++){x[i]=f(x[i-k]);}
  ```

  We or the compiler can **unroll** the loop in any way desired.

  If *n* − *m* were divisible by 2, consider unrolling the loop above with a dependency into groups of 2,

  ```
  for(i=m;i<n;i+=2){
          x[i ] =f(x[i-k ]);
          x[i+1]=f(x[i-k+1]);}
  ```

## Branching and conditional execution

- Example:

```
for(i=0;i<n;i++){
        if(e(a[i])>0.0){
                    c[i]=f(a[i]);
        } else {
                    c[i]=g(a[i]);
    }}
```

- Branch condition *e*(*x*) is usually simple but is likely to take a few cycles.
- The closest we can come to vectorizing this is to execute:
  - either both *f* and *g* and merge the results, or
  - alternatively, parts of both.

# Memory hierarchy

- Vector and superscalar processors have a two-level memory hierarchy:
  1. Small and fast register memory.
  2. Large and relatively slow main memory.
- This memory hierarchy is reflected in instruction sets and directly visible to the programmers.
- A simple actual memory hierarchy includes the following levels:
  1. Register memory
  2. Cache memory
  3. Main memory
  4. Disk memory
- The situation when a data item being referenced is not in the cache is called *cache miss*.
  - If the contribution of data transfer instructions into the total execution time of a program is substantial, a low no. of cache misses will significantly accelerate the execution of the program.
  - An obvious class of programs suitable for such optimization includes programs intensively using basic operations on arrays.

# Loop tiling

- Specific optimization performed by optimizing C compilers in order to minimize the number of cache misses
- Example: Consider the following loop nest:

```
for(i=0; i<m; i++) /* loop 1 */
        for(j=0; j<n; j++) /* loop 2 */
                if(i==0) b[j]=a[i][j]; else b[j]+=a[i][j];
```

  which computes the sum of rows of the *m x n* matrix *a*
- Remarks:
  - Data items accessed by reference b[j] are repeatedly used by successive iterations of loop 1.
  - If *n* of iterations of loop 2 is large enough, the data items may be flushed from the cache by the moment of their repeated use.
  - To minimize the flushing of repeatedly used data items, no. iterations of loop 2 may decrease
  - To keep the total no.iterations of this loop nest unchanged, a controlling loop is introduced.
- As a result the transformed loop nest looks as follows:

```
for(k=0; k<n; k+=T) /* additional controlling loop 0 */
        for(i=0; i<m; i++) /* loop 1 */
                for(j=k; j<min(k+T,n); j++) /* loop 2 */
                        if(i==0) b[j]=a[i][j];  else b[j]+a[i][j];
```

  This transformation is called tiling, and T is the tile size.
- Loop tiling improves temporal locality of nested loops by decreasing the no. iterations between repeatedly used array elements.