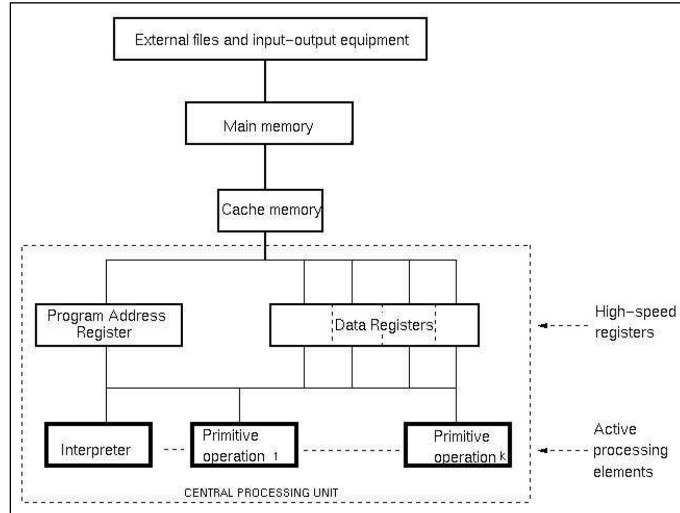
Traditional processor design



Ways to speed up execution

- To increase processor speed - Increase functionality of processor - add more complex instructions (CISC - Complex Instruction Set Computers)
- Need more cycles to execute instruction:
  - HAC430 assumes 2 cycles:
    - fetch instruction
    - execute instruction

How many:

- fetch instruction
- decode operands
- fetch operand registers
- decode instruction
- perform operation
- decode resulting address
- move data to store register
- store result

$\Rightarrow$ 8 cycles per instruction, not 2

Alternative - RISC

- Have simple instructions, each executes in one cycle
  - RISC - Reduced instruction set computer
- Speed - one cycle for each operation, but more operations.

For example: A=B+C

CISC: 3 instructions

Load Register 1, B
Add Register 1, C
Store Register 1, A

RISC:10 instruction

Address of B in read register
Read B
Move data Register 1
Address of C in read register
Read C
Move data Register 2
Add Register 1, Register 2, Register 3
Move Register 3 to write register
Address of A in write register
Write A to memory

Aspects of RISC design

- Single cycle instructions
- Large control memory - often more than 100 registers.
- Fast procedure invocation - activation record invocation part of hardware. Put activation records totally within registers.
- Implications:
  - Cannot compare processor speeds of a RISC and CISC processor:
  - CISC - perhaps 8-10 cycles per instruction
  - RISC - 1 cycle per instruction
  - CISC can do some of these operations in parallel.

Pipeline architecture

CISC design:

1. Retrieve instruction from main memory.
2. Decode operation field, source data, and destination data.
3. Get source data for operation.
4. Perform operation.

Pipeline design:

while Instruction 1 is executing
Instruction 2 is retrieving source data
Instruction 3 is being decoded
Instruction 4 is being retrieved from memory.

- Four instructions at once, with an instruction completion each cycle.

Impact on language design

- With a standard CISC design, the statement E=A+B+C+D will have the postfix EAB+C+D+= and will execute as follows:
  1. Add A to B, giving sum.
  2. Add C to sum.
  3. Add D to sum.
  4. Store sum in E.
- But, Instruction 2 cannot retrieve sum (the result of adding A to B until the previous instruction stores that result. This causes the processor to wait a cycle on Instruction 2 until Instruction 1 completes its execution.
- A more intelligent translator would develop the postfix EAB+CD++=, which would allow A+B to be computed in parallel with C+D

Further optimization

E=A+B+C+D

J=F+G+H+I

has a modified postfix of

EFAB+FG+CD+HI+++==

- In this case, each statement executes with no interference from the other, and the processor executes at the full speed of the pipeline

Conditionals

```
A = B + C;
if D then E = 1
    else E = 2
```

- Consider the above program. A pipeline architecture may even start to execute (E=1) before it evaluates to see if D is true.
- Options could be:
  - If branch is take, wait for pipeline to empty. This slows down machine considerably at each branch
  - Simply execute the pipeline as is. The compiler has to make sure that if the branch is taken, there is nothing in the pipeline that can be affected.
- This puts a great burden on the compiler writer. Paradoxically the above program can be compiled and run more efficiently as if the following was written:

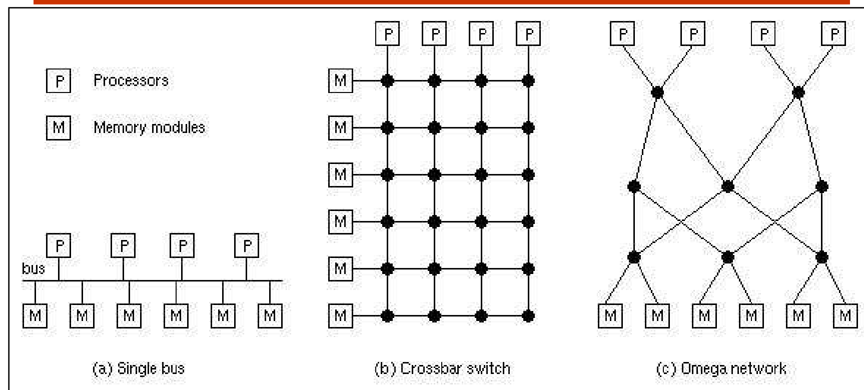
```
if D (A=B+C) then E = 1
          else E = 2
```

[Explain this strange behavior.]

Summary

- New processor designs are putting more emphasis on good language processor designs.
- Need for effective translation models to allow languages to take advantage of faster hardware.
- What's worse - simple translation strategies of the past, may even fail now.

Multiprocessor system architecture



Impact: Multiple processors independently executing

Need for more synchronization

Cache coherence: Data in local cache may not be up to date.

Tightly coupled systems

- Architecture on previous slide are examples of tightly coupled systems.
  - All processors have access to all memory
  - Semaphores can be used to coordinate communication among processors
  - Quick (nanosecond) access to all data
- Networks of machines (Loosely couples machines):
  - Processors have access to only local memory
  - Semaphores cannot be used
  - Relatively slow (millisecond) access to some data
  - Model necessary for networks like the Internet

Instruction scheduling

- Motivation
  - instruction latency (pipelining) - several cycles to complete instruction
  - instruction-level parallelism (VLIW, superscalar) - execute multiple instructions per cycle
- Issues
  - reorder instructions to reduce execution time
  - static schedule → insert NOPs
  - dynamic schedule → pipeline stalls
  - preserve correctness
  - operate efficiently
  - interactions with optimizations
- Sources of latency (hazards)
  - data - operands depend on previous instruction
  - structural - limited hardware resources
  - control - targets of conditional branches

Approach

- schedule after register allocation (postpass) - model used to estimate execution time
- A legal schedule
  assume each $i \in$ *Instr* has delay(i)
  a legal schedule S maps each $i \in$ *Instr* onto a
  non-negative integer representing its cycle number
  (i.e., time instruction is executed)
  if i2 "depends" on i1, $S(i1) + delay(i1) \leq S(i2)$
  there are no more instructions in any cycle than can be issued by
  the machine the length of schedule S, denoted L(S), is

$$L(S) = MAX_{i \in Instr} (S(i) + delay(i))$$

  S is optimal if $L(S) \leq L(T)$, $\forall$ legal schedules T
- Scope
  - basic blocks (list scheduling)
  - branches (trace scheduling, percolation)
  - loops (unrolling, software pipelining)

Data dependencies

- Dependences → memory locations instead of values
- Statement b depends on statement a if there exists:
  - true or flow dependence - a writes a location that b later reads (read-after-write or RAW)
  - anti-dependence - a reads a location b later writes (write-after-read or WAR)
  - output dependence - a writes a location that b later writes (write-after-write or WAW)
- Another dependence (doesn't constrain ordering)
  - input dependence - a reads a location that b later reads (read-after-read or RAR)
- Example

true	anti	output	input
a =	= a	a =	= a
= a	a =	a =	= a

Precedence graph

- Construction
  - > instructions → nodes
  - > dependences → edges
- Example

```
1   load   r1, X
2   load   r2, Y
3   mult   r3, r2, r1
4   load   r4, A
5   mult   r5, r4, r3
6   add    r6, r2, r3
7   mult   r2, r5, r6
8   load   r8, B
9   add    r9, r8, r2
```

- Register renaming eliminates anti and output dependenciess

```
a =      a =
 = a      = a
a =      b =
 = a      = b
```


List scheduling

- Algorithm
  - > rename to eliminate anti/output dependenciess
  - > construct precedence graph
  - > assign priorities to instructions
  - > iteratively select & schedule instructions
- candidates ← roots of graph
  while candidates remaining
  pick highest priority candidate
  schedule instruction
  add exposed instructions to candidates
- Two flavors of list scheduling

Forward list scheduling	Backward list scheduling
start with available ops	start with no successors
work forward	work backward
ready → all ops available	ready → latency covers uses

Scheduling heuristics

- Problems
  - how to choose between ready instructions?
  - NP-hard for straight-line code
- Heuristics used to prioritize candidates
  - ready to execute (no stalls)
  - highest latency (more overlap)
  - most immediate successors (create candidates)
  - most descendants (create more candidates)
  - longest weighted path to root (critical path)
- Approach
  - use multiple heuristics (help break ties)
  - try multiple schedules (take best result)

Scheduling example

- Machine model
  - 3-cycle latency for load
  - 2-cycle latency for mult
  - 1-cycle latency for add
- Example

1	load	r1, X
2	load	r2, Y
3	mult	r3, r2, r1
4	load	r4, A
5	mult	r5, r4, r3
6	add	r6, r2, r3

Cycle	1	2	3	4	5	6	7	8
Candidates								
Forward scheduling								
Candidates								
Backward scheduling								
Candidates								
2-instruction scheduling								

Trace scheduling

- Overview- a *trace* is a path through code
  - examine branch probabilities
  - find trace representing most likely path
  - schedule instructions for trace
  - emit repair code around trace
  - repeat as necessary
- Example

```
    code1                 code1
code2  code4              code2  code4
code3  code5              code3  code5
    code6                 code6
```

- Result
  - better instruction schedule along trace
  - less efficient schedule off trace
  - increase in code size (from repair code)

Trace repair code

- Code moved below split
  - create basic block B at branch target
  - replicate code C moved below split
  - insert C in B in original order
- Example

```
   code1                       if (...)
   if (...)                 code1    code1
                            code2    code3
code2  code3
```

- Code moved above split
  - only move code which is dead off trace
  - may perform unnecessary work
- Example

```
  code1            code1
  if (...)         code2
                   if (...)
code2  code3                 // code2 is dead
                   code3
```

Loop unrolling

- Approach
  - create multiple copies of loop
  - more candidates for scheduling
- Example

```
   do i=1,N                  do i=1,N,3
1     load                   load
2     mult                   mult  load
3                                  mult  load
4     store                  store       mult
5                                  store
6                                        store

    [1 pass=4 cycles]        [3 passes= 6 cycles]
```

- Problems
  - choosing degree of unrolling k
  - pipeline hiccup every k iterations
  - increased compilation time
  - instruction cache overflow

Software pipelining

- Approach
  - overlap iterations of loop
  - select schedule for loop body
  - initiate iteration after every k cycles, before previous iterations complete
  - constant initiation interval
- Example

```
T     i=1       i=2       i=3       i=4
1     load
2     mult      load
3               mult      load
4  L: store               mult      load  (cjmp L)
5               store               mult
6                         store
7                                   store
```

- Properties
  - prolog, epilog code for pipeline
  - steady state within body of loop
  - hiccups in pipeline at entry, exit

Branch prediction

- Will a conditional branch be taken?
  - affects instruction scheduling
  - execution penalty for incorrect guess
- Prediction approaches
  - hardware history (branch bit)
  - history plus branch correlation
  - profiling (feedback to compiler)
  - compile-time heuristics
- Static branch prediction
  - no run-time information
  - single prediction for all executions
  - perfect prediction = 50--100% correct
- Simple (target) heuristic
  - predict conditional branch taken
  - catches loop back edges

Branch prediction

- Loop branch heuristic
  - find forward, back, exit edges
  - predict back edge, non-exit edge
- Op code heuristic
  - predict greater than zero (error conditions)
  - predict floating point values differ
- Loop heuristic - predict branch leading to loop header
- Call heuristic - predict branch not leading to call
- Return heuristic - predict branch not leading to return
- Guard heuristic - predict branch leading to guarded variable
- Store heuristic - predict branch leading to store of variable
- Pointer heuristic - predict pointer not NULL, pointers differ

Compiling high performance machines

- Issues
  - Parallelism
  - locality
- Classical compilation
  - focus on individual operations
  - scalar variables
  - flow of values
  - unstructured code
- High-performance compilation
  - focus on aggregate operations (loops)
  - array variables
  - memory access patterns
  - structured code
- Issues
  - dependence analysis
  - loop transformations

Forms of parallelism

- Instruction-level parallelism
  - for superscalar and VLIW architectures
  - examine dependences between statements
  - very fine grain parallelism

```
A = 1        B = C
```

- Task-level parallelism
  - for multiprocessors
  - examine dependences between tasks
  - parallelism is not scalable

```
do i = 1,10          do i = 1,10
  A(i) = A(i+1)        B(i) = B(i+1)
```

- Loop-level parallelism
  - for vector machines and multiprocessors
  - examine dependences between loop iterations
  - parallelism is scalable

```
doall i = 1,10
  A(i) = B(i+1)
```

Loop-level parallelism

- Basic approach
  - execute loop iterations in parallel
  - safe if no loop-carried data dependences (i.e., no accesses to same memory location)

```
do i = 1,10          doall i = 1,10
  A(i) = A(i+1)        A(i) = A(i+10)
```

- Several parallel architectures
  - vector processors
    ```
    A[1:10] = B[2:11]
    ```
  - multiprocessors
    ```
    doall i = 1,10
    A(i) = B(i+1)
    ```
  - message-passing machines
    ```
    if (...) send B(1)
    if (...) recv B(11)
    do i = L,B
      A(i) = B(i+1)
    ```


Exposing parallelism

Scalar analysis
- improve precision of dependence tests
- eliminate unnecessary scalar statements
- based on data-flow analysis

Solution techniques
- forward propagation (expose value of scalar variables)

```
do i = 1,10       do i =1,10
   k = i+1           k = i+1
   A(k) =            A(i+1) =
```

- constant propagation

```
k = 1           k = 1
do i = 1,10     do i =1,10
  A(i+k) =         A(i+1) =
```

- induction variable recognition

```
k = 1             k = 1
do i = 1,10       do i =1,10
   k = k+1           A(i+1) =
   A(k) =         k = 11
```

Vectorization

- Vector processors
  - Operations on vectors of data
  - Overlap iterations of inner loop

```
doall i=1,10           A[1:10] = 1.0
  A(I) = 1.0 B[1:10] = A[1:10]
  B(i) = A(i)
```

  - Exploits fine-grain parallelism
  - Expressed in vector languages (APL, Fortran 90)
- Execution model
  - Single thread of control
  - Single instruction, multiple data (SIMD)
  - Load data into vector registers
  - Efficiently execute pipelined operations
- Issues
  - Vector length – coalesce loops to reduce overhead
  - Control flow – convert conditions into explicit data
  - Not very popular today!

Parallelization

- Multiprocessors
  - Multiple independent processors (MIMD)
  - Assign iterations to different processors

```
doall j=1,10            do j=l,u
  do i=1,10               do i=1,10
    A(I,j) = 1.0            A(i,j)=1.0
```

  - Exploits coarse grained parallelism
- Execution model
  - Fork-join parallelism
  - Master executes sequential code
  - Workers (and master) execute parallel code
  - Master continues after workers finish
- Issues
  - Granularity – larger computation partitions to reduce overhead
  - Scheduling – policy for assigning iterations to processors

multithreading

- High latency events
  - I/O
  - Interprocess communication
  - Page miss
  - Cache miss
- Multiple threads of execution
  - Switch to new thread after event
  - Overlaps computation with event
  - Requires threads, efficient context switch
- Hardware support
  - Switch sets of registers
  - Shared caches
- Software support
  - Uncovers parallelism for multiple threads
  - Reduce context switch overhead