Superscalar Processors

Superscalar Processors

- Goal is to execute an arbitrary instruction stream and to achieve a clocks per instruction (CPI) of < 1.0
  - Usually, all instruction streams are ‘legal’ since the instruction set architecture may have been previously implemented on non-superscalar architectures and thus legacy codes must be executed.
    - If the Instruction Set Architecture (ISA) has been designed from the beginning to be implemented on a super scalar architecture, then have the freedom of defining what a legal code stream is.
  - Current state of the art Superscalar processors implement 2-4 instructions per clock

Examples of Different Implementations of Same ISA

- 80x86 ISA defined in 1978
- Implementations are:
  - 8086, 8086, 80186, 80286, 80386: Multicycle implementations; each successive implementation widened external data busses, increased external address space, reduced clocks per instruction
  - 80486 : Pipelined implementation, also integrated Floating point unit on same die
  - Pentium: Superscalar implementation, 2 instructions max per clock
  - iA64 (Merced)? : > 2 instructions per clock

Figure 7.2, 7.3 of Text (pgs. 188, 189)

- Show evolution of current superscalar processors
- Interesting differences between CISC ISAs (x86, 68000) and RISC ISAs
  - RISC ISAs have higher instructions/clock than CISC ISAs because of lower complexity
  - RISC ISAs had superscalar implementations before CISC ISAs.
  - Text mentions that in some implementations, CISC instructions are converted on-the-fly to multiple RISC instructions for more efficient processing (AMD K5 implementation of x86 ISA).

Tasks of Superscalar Processing

- Parallel Decode/ Instruction Issue
  - Pre-decoding
  - Shelving buffers, Reservation stations, register renaming
- Execution
- Preserving sequential consistency
  - Reorder buffer
- Preserving sequential consistency within exception processing

Parallel Decode/Instruction Issue

- Need to determine for multiple instructions:
  - Are the operands available, and if so, fetch them
  - Which execution units do the instructions go to (choose from multiple execution units)
- Superscalar implementations may use multiple stages for decode/issue
  - PPC 601, PPC 604, UltraSparc take cycles, Alpha 21064 takes 3 cycles, Pentium Pro requires 4.5 cycles

Predecoding

- Do some pre-decoding as part of Instruction cache fill
  - Attaches extra bits to each instruction
  - Bits denominate the instruction class, type of resources required, and even calculation of branch target address
  - Number of extra bits range from 4 to 7 (table 7.1)
- Power PC 620 and R10000 use predecoding
  - Only one cycle for decoding and issue
- Some modern processors don't use predecoding
  - Alpha 21116 and PentiumPro

Instruction Issue

- Blocking (Direct) issue versus Shelved (Indirect) Issue
- Blocking Issue will wait until operands are available (either in register file or thru forwarding paths) before issuing the instruction to the Execution Unit
- Shelved (Indirect) Issue sends instructions to a ‘shelving’ station attached to the Execution Unit
  - If operands are not available yet, a register tag is written instead
  - Execution unit will get its operands from the shelving buffer when it becomes available.
  - Instruction issue not blocked if operands not available, only blocked if shelving buffers are full!

More on Shelving (Figure 7.11)

- Shelving buffers also called

  Reservation Stations (RS)
- Shelving decouples instruction issue from dependency checking
  - Dependencies are resolved when instruction in shelving buffer is dispatched to an Execution Unit
- Shelving buffer organization (Figure 7.25)
  - One buffer per Execution Unit
  - One buffer serves a group of Execution Units
  - One buffer serves all Execution Units (Central Reservation Station)

How many entries in Shelving Buffers ?

- Table 7.2 compares number of shelves for different processors
- Will be dependent upon number of Execution Units, and whether or not individual, group, or a central scheme is used for shelving buffers
- A Central reservation station is the most efficient in terms of number of entries, but most complex in terms of number of read/write ports necessary to access entries.

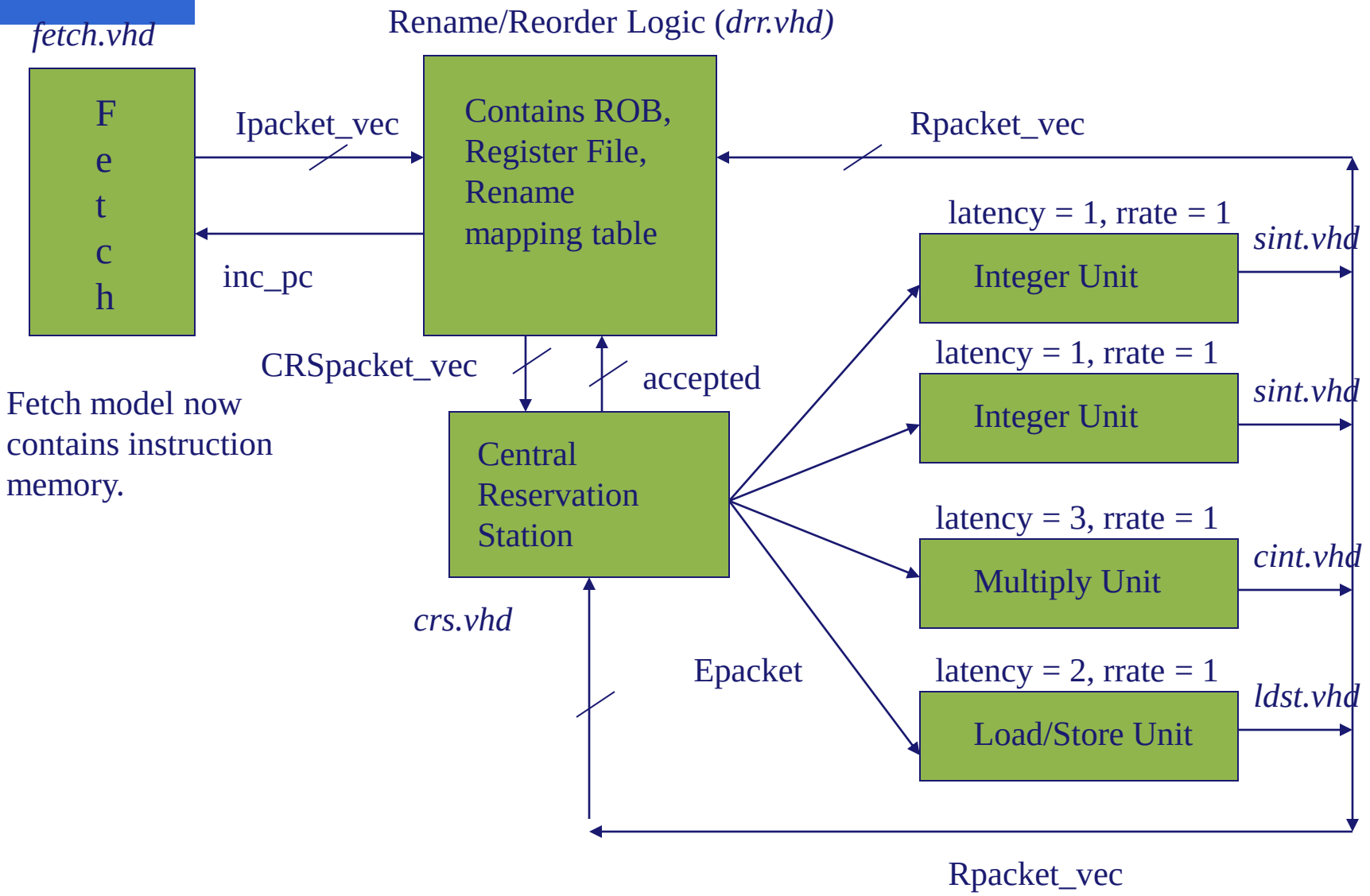
Shelving Buffer Entry (Figure 7.38)

- A Shelving buffer entry contains:
  - Source operand value
  - Source operand tags (register numbers)
  - Operation Code
  - Source Operand Valid bits
  - Destination Register Number
- Valid bits indicate if source operands are ready or not
- Forwarded results from Execution Units are written to shelving buffer
- When operands ready, instruction dispatched to execution unit.

Superscalar Processors 2

3/27/2012

SuperScalar Abstract Machine



Operations

- All control logic concentrated in Central Reservation Station (CRS) and Decode/Rename/Reorder (DRR) block
- Will divide operations into high/low clock phases
  - Data moved between blocks on rising clock edge
  - High clock phase used for updates (writes)
  - Low clock phase used for reads

Central Reservation Station (CRS)

- High Clock Phase
  - Accept issued instructions from DRR
    - Will use status bits to indicate which instructions were accepted.
    - One status bit for each instruction
  - If instruction is a NOP, accept it, but do NOT place it in a shelf
    - Will not actually execute the NOP.
    - NOPs do not consume resources

Central Reservation Station (CRS) cont.

- Low Clock Phase
  - Update all shelves with values from result bus (output of execution units)
    - Entries which are updated can be dispatched this clock
    - In reality, this update is being done throughout the high phase and into the low phase, but due to modeling constraints, will do it in low phase only.
  - Dispatch ready instructions to execution units
    - Examine all shelves for operands with both valid bits set
    - Dispatching the instructions frees the shelf entry (mark it as invalid)
    - Always dispatch EUMAX instructions - dispatch a NOP if can't fine a shelf entry that is valid.

Decode/Rename/Reorder Block (DRR)

- High Clock Phase
  - Will do ALL work in low clock phase to make modeling easier.
- Low Clock Phase
  - Retire Finished instructions
    - Search Reorder Buffer (ROB); any entry that is marked as finished, write result to register file
    - Note that values written to register file in this high clock can be read in the next low clock phase (flow thru register file).
  - Update ROB with values from result bus
    - An entry that is updated (marked as finished) cannot be retired in this clock cycle, will be retired next clock cycle

Decode/Rename/Reorder Block (DRR) cont.

- Low Clock Phase (cont)
  - Check if all instructions from last issue were accepted by CRS
    - if not, replace accepted instructions with NOPS and re-issue same values
  - Check if room in ROB
    - if not, issue all NOPs to CRS and don't increment PC
  - At this point, can accept all instructions from Fetch block; tell Fetch Block to increment PC, place four new instructions in ROB
  - Fetch operands (if available) for each of the 4 instructions
    - If not available, then use 'renamed' register number as placeholder
    - Note that this requires DECODING!
    - Operand Fetch MUST BE DONE BEFORE renaming because a source and destination may be same register!
  - Rename destination registers for 4 instructions from
    - Renaming is simply allocating entries in ROB
  - Issue instructions to CRS

VHDL Model

- You will need to fill in the internals of the crs.vhd (Central Reservation Station entity) and drr.vhd (Decode/Rename/Reorder entity) files
  - For the first assignment, only need to execute instruction sequences that contain integer add (addreg, addimm) and multiply (multreg, multimm) instructions.
- The fetch block now contains the instruction memory and reads instructions from a file
  - The default file name is ‘program.txt’ which should reside in the directory from where you execute the model.
  - Can use symbolic links to point this file at different test sequences.

Sample test file

Comment line

Debug Flag

```
# Test basic dependency code
#
ADDIMM 8 0 0 25 FALSE   -- add r8,r0,#25  ;
ADDIMM 8 8 0 30  FALSE  -- add r8,r8,#30  ;
ADDIMM 9 8 0 50  FALSE  -- add r9,r8,#50  ;
ADDREG 10 9 8 0   TRUE  -- add r10,r9,r8  ;
```

Dest Reg, Src1 Reg, Src2 Reg, Immediate Field

Instruction, case sensitive, name same as VHDL enumerated type names for 'ops' type found in *abmtypes.vhd* .

More on debug flag

Goal: Would like to print register contents after a particular instruction is retired for debugging reasons.

Each packet type (epacket, rpacket, etc) defined in *abmtypes.vhd* has a boolean field called pflag. This flag is initially set by the fetch block after parsing the line for that instruction. This flag should be passed unchanged by each entity. When the instruction is RETIRED, if pflag is TRUE, then the *ddr* entity should set the shared variable *printreg* to TRUE. This will cause the registers to be printed and the *printreg* variable to be reset back to FALSE.

The shared variable printreg is defined in *abmtypes.vhd*; the code for printing the registers is already contained in the *drr.vhd* entity.

Record types in *abmtypes.vhd*

- ipacket - defines an instruction (opcode, pflag, dest, src1, src2, immv). Passed from *fetch* to *drr*.
- CRSpacket - defines an issued instruction (opcode, pflag, dest, src1, src1_valid, src2, src2_valid, immv, valid). Passed from *drr* to *crs*.
  - valid flag only used within CRS to mark shelf entry as free or empty.
- Epacket - defines a dispatched instruction (opcode, pflag, dest, src1, src2, immv). Passed from *crs* to execution units.

Record types in *abmtypes.vhd* (cont.)

- Rpacket - output of execution units (dest, dest_v, valid, pflag). Passed from execution units to *crs* and *drr*.
  - valid flag is set to FALSE if this was the result a NOP. The crs and drr should ignore any Rpackets with valid flag set to FALSE
- ROBpacket - defines a Reorder buffer entry (dest, dest_v, status, pflag)
  - Used internally by *drr* - Reorder buffer is array of these records
- MAPpacket - defines a Reorder buffer map entry (ptr, valid)
  - Used internally by drr - Reorder buffer map is array of these records.

Important constants in *abmtypes.vhd*

- EUMAX - total number of execution units
- SHELFMAX - total number of shelves for integer units
- LS_SHELFMAX - number of shelves for Load/Store unit
- REGMAX - number of registers
- ROBMAX - number of entries in the Reorder buffer
- Use these constants in your code!!! Do NOT use hardcoded numbers! Will want to change the values for these constants later!

Status Messages

If you detect that the ROB is full, print out a message:

```
if (ROB is full) then
  assert false
    report LF&"ROB Full" &LF
  severity note;
end if;
```

If you detect that the CRS did not accept all instructions, print out a message:

```
if (not all accepted) then
  assert false
    report LF&"All instructions not accepted by CRS"&LF
   severity note;
end if;
```

Debugging, Testing

Test your code with different code sequences! I have given you some sample code sequences, but you should test your code with more than these.

If you have trouble, feel free to see me - be ready to execute your code and have debug spots picked out that you want me to look at.

The next simulation will also involve this model - if you don’t get this model working, then you will not be able to do the next simulation.

Pipelined Processors

Definitions

- *FX pipeline* - Fixed point pipeline (integer pipeline)
- *FP pipeline* - Floating Point pipeline
- *Cycle time* - length of clock period for pipeline, determined by slowest stage.
- *Latency* used in referenced to RAW hazards - the amount of time that a result of a particular instruction takes to become available in the pipeline for a **subsequent** dependent instruction (measured in multiples of clock cycles)

RAW Dependency, Latencies

- **Define-use Latency** is the time delay after decoding and issue of an instruction until the result becomes available for a subsequent RAW dependent instruction.

  add r1, r2,r3

  add r5, r1, r6 ⟸ define-use dependency

  Usually one cycle for simple instructions.

- **Define-use Delay** of an instruction is the time a subsequent RAW-dependent instruction has to be stalled in the pipeline. It is one less cycle than the *define-use latency*.

RAW Dependency, Latencies (cont)

- If define-use latency = 1, then define-use delay is 0 and the pipeline is not stalled.
  - This is the case for most simple instructions in the FX pipeline
  - Non-pipelined FP operations can have define-use latencies from a few cycles to a 10's of cycles.
- Load-use dependency, Load-use latency, load-use delay refer to load instructions

```
load r1,  4(r2)
add   r3, r1, r2
```

Definitions are the same as *define-use dependency*, l*atency*, and *delay*.

More Definitions

Repetition Rate R (throughput) - shortest possible time interval between subsequent independent instructions in the pipeline

Performance Potential of a Pipeline - the number of independent instructions which can be executed in a unit interval of time:

$$P = 1 \ / \ (R * t_c)$$

R: repetition rate in clock cycles

t_c : cycle time of the pipeline

Table 5.1 from Text (latency/repetition rate)

Processor	CycleTime	Prec	Fadd	FMult	FDiv	Fsqrt
a21064	7/5/2	s	6/1	6/1	34	-
		p	6/1	6/1	63	-
Pentium 6/5/3.3	s	3/1	3/1	39	70	
	d	3/1	3/1	30	70	
Pentium Pro	6.7/5/3.3s	3/1	5/2	18	29	
		d	3/1	5/2		
HP PA 8000	5.6	s	3/1	3/1	17	17
		d	3/1	3/1	31	31
SuperSparc	20/17	s	1/1	3/1	6/4	8/6
		d			9/7	12/10

How many stages?

- The more stages, the less combinational logic within a stage, the higher the possible clock frequency
  - More stages can complicate control. Dec Alpha has 7 stages for FX instructions, and these instructions have a define-use delay of one cycle for even basic FX instructions
  - Becomes difficult to divide up logic evenly between stages
  - Clock skew between stages becomes more difficult
  - Diminishing returns as stages become large
- *Superpipelining* is a term used for processors that use a high number of stages.

Dedicated Pipelines versus Multifunctional Pipelines

- Trend in current high performance CPUs is to used different logical AND physical pipelines for different instruction classes
  - FX pipeline (integer)
  - FP pipeline (floating point)
  - L/S pipeline (Load/Store)
  - B pipeline (Branch)
- Allows more concurrency, more optimization
  - Silicon area more plentiful

Sequential Consistency

- With multiple pipelines, how do we maintain sequential consistency when instructions are finishing at different times?
  - With just two pipelines (FX and FP), we can lengthen the shorter pipeline with statically or dynamically. Dynamic lengthening would be used only when hazards are detected.
  - We can force the pipelines to write to a special unit called a Renaming Buffer or Reordering Buffer. It is the job of this unit to maintain sequential consistency. Will look at this in detail in Chapter 7 (superscalar).

RISC versus CISC pipelines

- Pipelines for CISC are required to handle complex memory to register addressing
  - mov r4, (r3, r2)4 ⟸ EA is r3 + r2 + 4
  - Will have an extra stage for Effective address calculation (see Figures 5.40, 5.41, 5.43)
  - Some CISC pipelines avoid a load-use delay penalty (Fig 5.54, 5.56)
- RISC pipelines have a load-use penalty of at least one
- Determining load-use penalties when multiple pipelines are in action are instruction sequence dependent (ie., 1, 2, more than 2 cycles)

Some other important Figures in Chapter 5

- Figure 5.26 (illustrates use of both clock phases for performing pipeline tasks)
- Figure 5.31, Figure 5.32 (Pentium Pipeline, shows difference between logical and physical pipelines)
- Figure 5.33, Figure 5.34 (PowerPC 604 - first look at a modern superscalar processor)

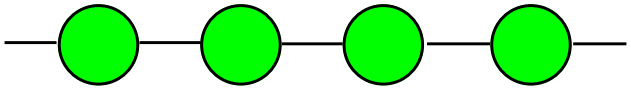
IMPORTANT NOTE

Review Lecture on Optimum pipeline equation and deduction!

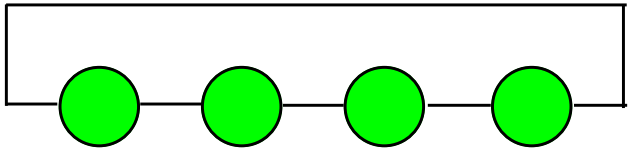
Review Text on Shelving, Issue/dispatch, ROB, in-order and out-of-order issue, fixed and gliding windows….

Important questions can arise here!

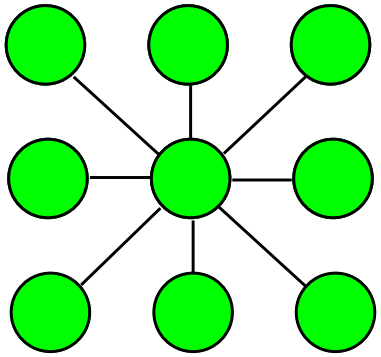
Parallel Architectures



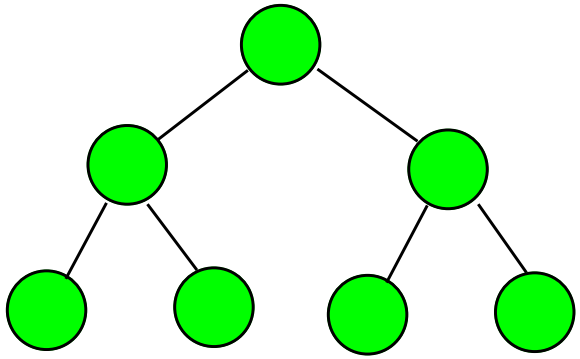
Linear array



Ring



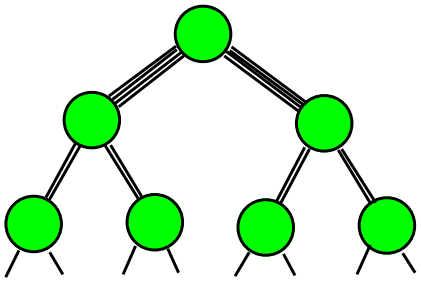
Star



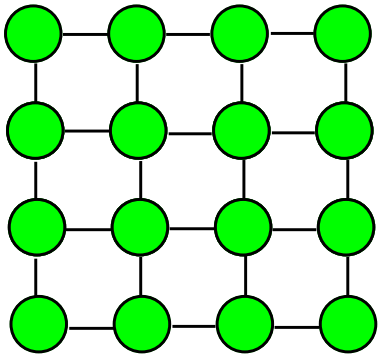
Binary tree

MAIN NETWORK TOPOLOGIES (a)

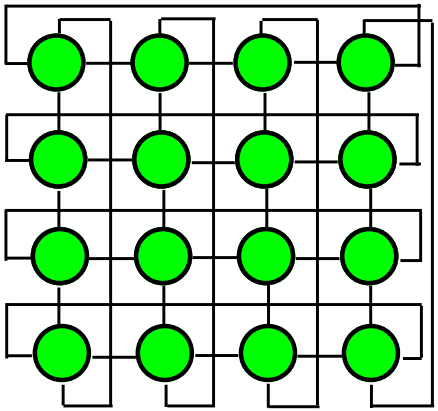
Gyula Fehér feher@novserv.obuda.kando.hu



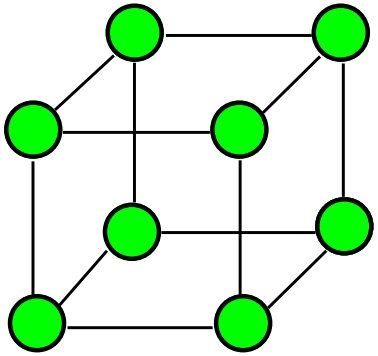
Fat tree



2-D mesh



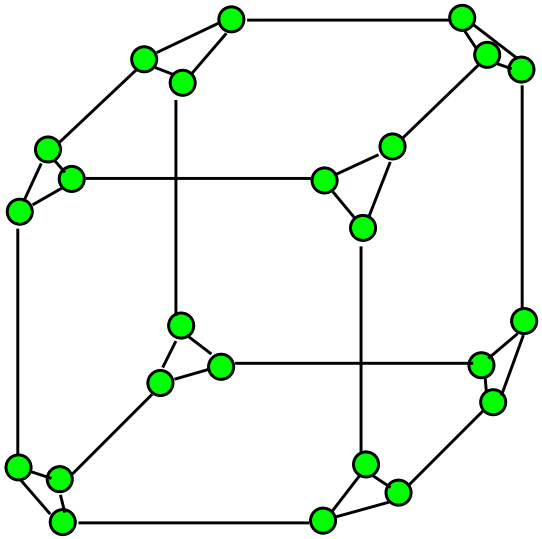
2-D wraparound mesh



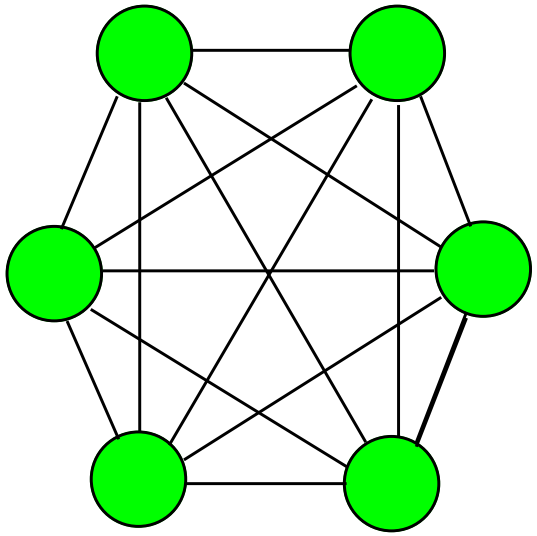
2-D mesh

MAIN NETWORK TOPOLOGIES (b)

Gyula Fehér feher@novserv.obuda.kando.hu



3-cube connected cycle



Completely connected

MAIN NETWORK TOPOLOGIES (c)

Gyula Fehér feher@novserv.obuda.kando.hu

Topology	*Node degree*	*Diameter*	*Bisection width*	*Arc connectivity*	*Cost*
Linear array	1 or 2	N-1	1	1	N-1
Ring	2	N/2	2	2	N
Star	1 or N-1	2	1	1	N-1
Binary tree	1, 2 or 3	2log((N+1)/2)	1	1	N-1
2-D mesh	2, 3 or 4	$2(N^{1/2}-1)$	$N^{1/2}$	2	$2(N-N^{1/2})$
2-D wraparound mesh	4	$N^{1/2}$	$2N^{1/2}$	4	2N
3-D cube	3, 4, 5 or 6	$3(N^{1/3}-1)$	$N^{2/3}$	3	$2(N-N^{2/3})$
Hypercube	logN	logN	N/2	logN	(NlogN)/2
Completely connected	N-1	1	$N^2/4$	N-1	N(N-1)/2

Summary of static network parameters for the main topologies

Gyula Fehér feher@novserv.obuda.kando.hu