Principles of Computer Architecture

Miles Murdocca and Vincent Heuring

Chapter 10: Trends in Computer Architecture

Chapter Contents

Instruction Frequency

- **Frequency of occurrence of instruction types for a variety of languages. The percentages do not sum to 100 due to roundoff. (Adapted from Knuth, D. E., *An Empirical Study of FORTRAN Programs, Software—Practice and Experience*, 1, 105-133, 1971.)**

Statement	Average Percent of Time
Assignment	47
If	23
Call	15
Loop	6
Goto	3
Other	7

Complexity of Assignments

- **Percentages showing complexity of assignments and procedure calls. (Adapted from Tanenbaum, A., *Structured Computer Organization*, 4/e, Prentice Hall, Upper Saddle River, New Jersey, 1999.)**

	Percentage of Number of Terms in Assignments	Percentage of Number of Locals in Procedures	Percentage of Number of Parameters in Procedure Calls
0	–	22	41
1	80	17	19
2	15	20	15
3	3	14	9
4	2	8	7
≥ 5	0	20	8

Speedup and Efficiency

- **Speedup *S* is the ratio of the time needed to execute a program without an enhancement to the time required with an enhancement.**

$$S = \frac{T_{wo}}{T_w} \qquad S = \frac{T_{wo} - T_w}{T_w} \times 100$$

- **Time *T* is computed as the instruction count *IC* times the number of cycles per instruction *CPI* times the cycle time τ.**

$$T = IC \times CPI \times \tau$$

- **Substituting *T* into the speedup percentage calculation above yields:**

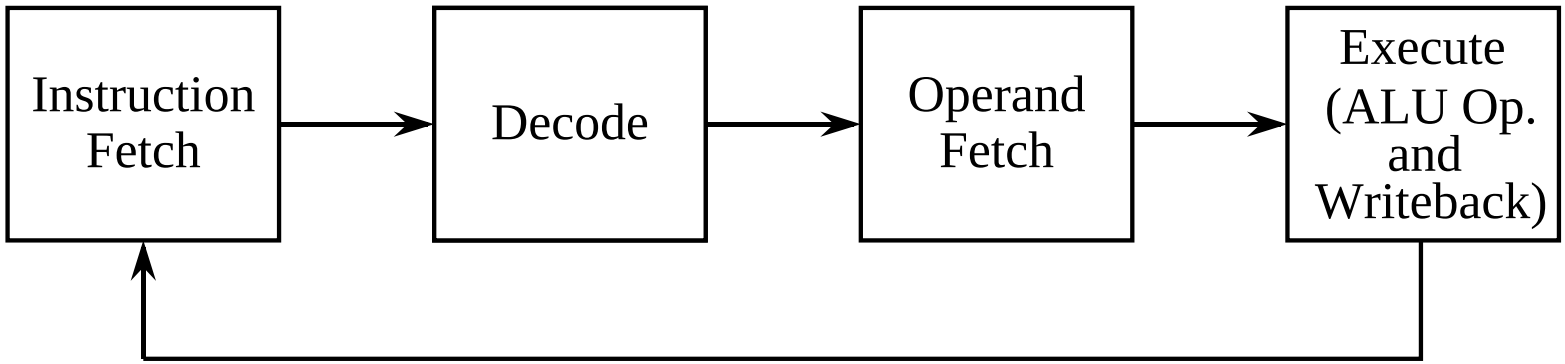
$$S = \frac{IC_{wo} \times CPI_{wo} \times \tau_{wo} - IC_w \times CPI_w \times \tau_w}{IC_w \times CPI_w \times \tau_w} \times 100$$

Example

- ***Example:*** **Estimate the speedup obtained by replacing a CPU having an average CPI of 5 with another CPU having an average CPI of 3.5, with the clock period increased from 100 ns to 120 ns.**
- **The previous equation becomes:**

$$S = \frac{5 \times 100 - 3.5 \times 120}{3.5 \times 120} \times 100 = 19\%$$

Four-Stage Instruction Pipeline



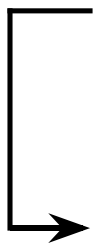
Pipeline Behavior

- **Pipeline behavior during a memory reference and during a branch.**

	Time 1	2	3	4	5	6	7	8
Instruction Fetch	addcc	ld	srl	subcc	be	nop	nop	nop
Decode		addcc	ld	srl	subcc	be	nop	nop
Operand Fetch			addcc	ld	srl	subcc	be	nop
Execute				addcc	ld	srl	subcc	be
Memory Reference						ld		

Filling the Load Delay Slot

- **SPARC code, (a) with a nop inserted, and (b) with `srl` migrated to nop position.**



```
srl   %r3, %r5
addcc %r1, 10, %r1
ld    %r1, %r2
nop
subcc %r2, %r4, %r4
be    label
```

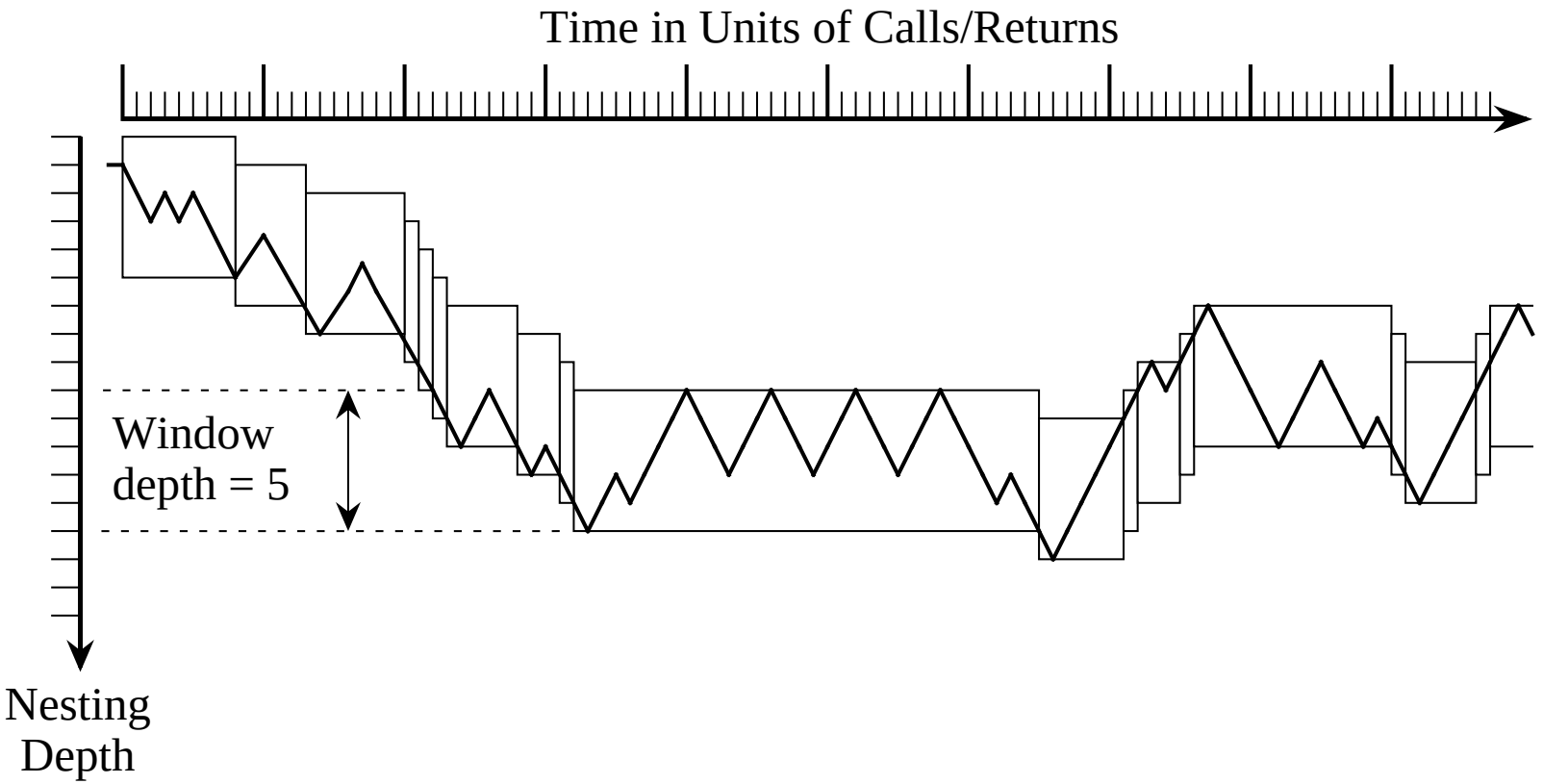
(a)

```
addcc %r1, 10, %r1
ld    %r1, %r2
srl   %r3, %r5
subcc %r2, %r4, %r4
be    label
```

(b)

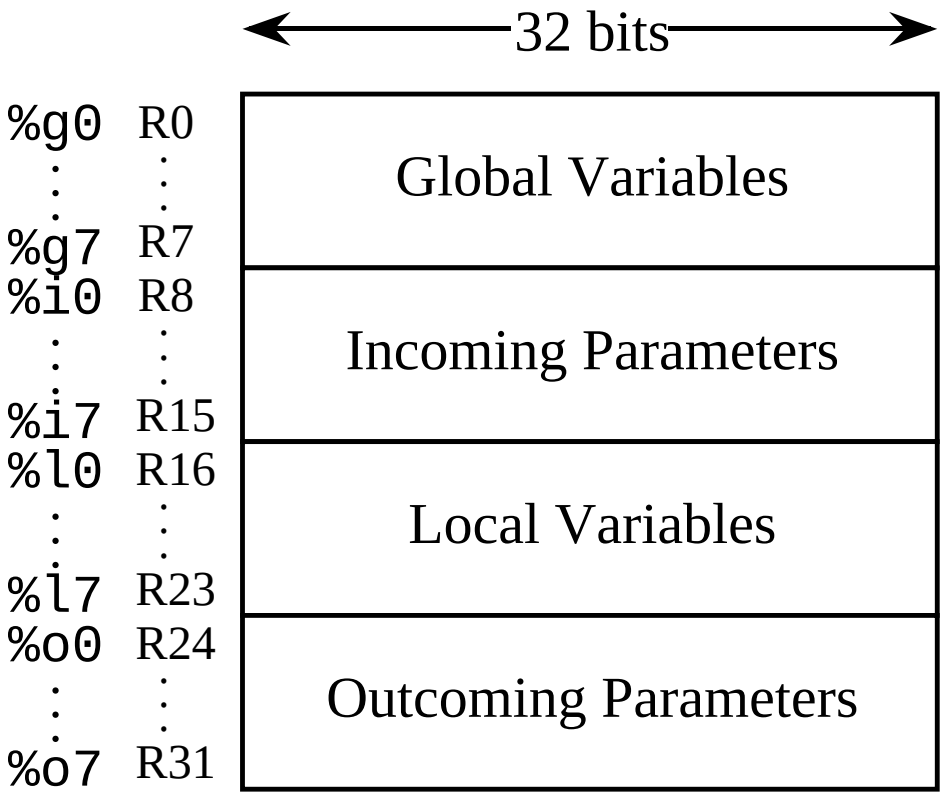
Call-Return Behavior

- **Call-return behavior as a function of nesting depth and time (Adapted from Stallings, W., *Computer Organization and Architecture: Designing for Performance*, 4/e, Prentice Hall, Upper Saddle River, 1996).**

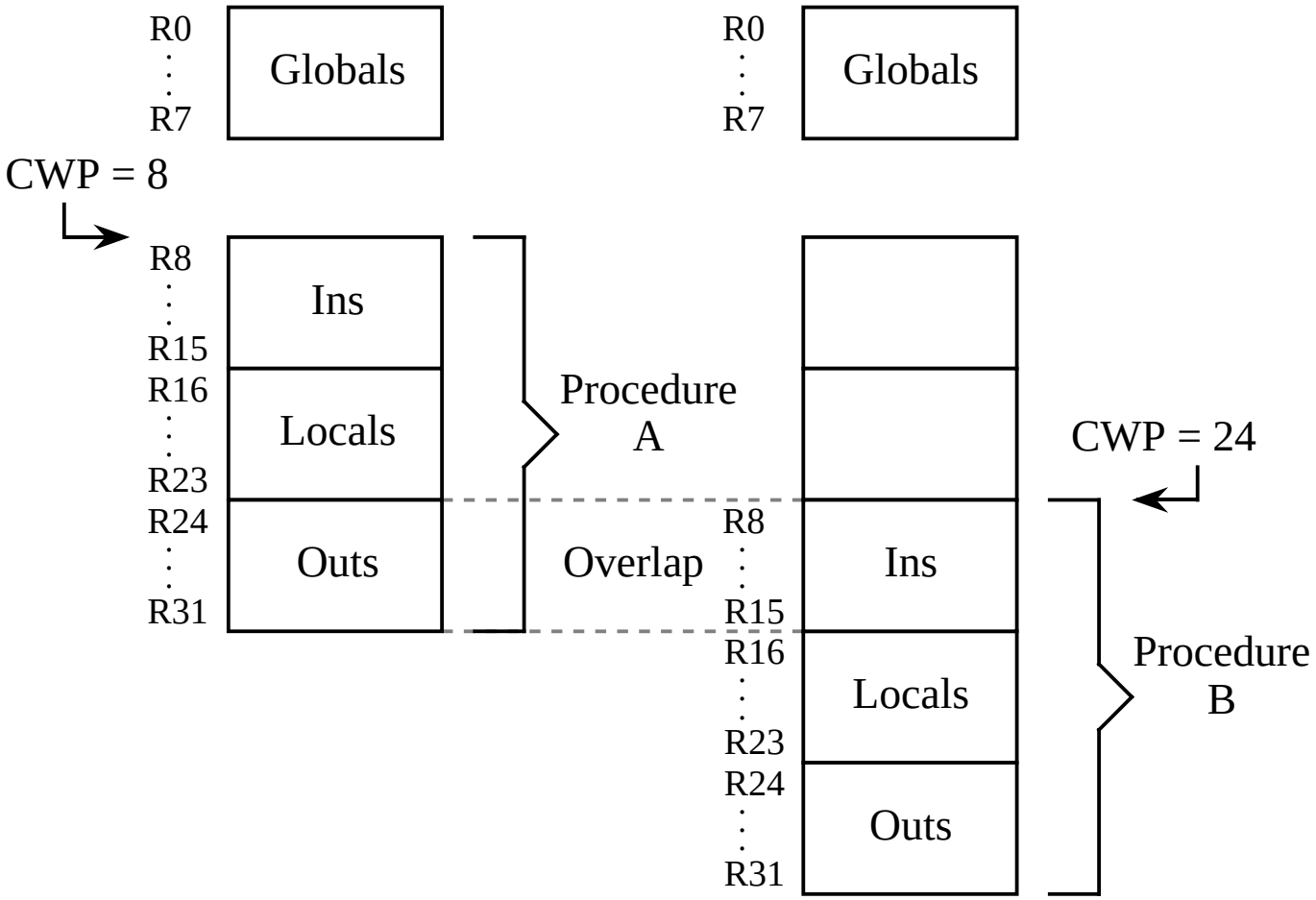


SPARC Registers

- **User view of RISC I registers.**



Overlapping Register Windows



Example: Compiled C Program

- **Source code for C program to be compiled with gcc.**

```
/* Example C program to be compiled with gcc */

#include
<stdio.h>
main ()
{
    int a, b, c;

    a = 10;
    b = 4;
    c = add_two(a, b);

    printf("c = %d\n", c);
}

int add_two(a,b)
int a, b;
{
    int result;

    result = a + b;
    return(result);
}
```

gcc Generated SPARC Code

```
! Output produced by gcc compiler on Solaris (Sun UNIX)
! Annotations added by author

.file   add.c   ! Identifies the source program
.section        ".rodata"       ! Read-only data for routine main
        .align 8        ! Align read-only data for routine main on an
                        ! 8-byte boundary
.LLC0
        .asciz  "c = %d\n"      ! This is the read-only data
.section        "text"  ! Executable code starts here
        .align 4  ! Align executable code on a 4-byte (word) boundary
        .global main
        .type   main,#function
        .proc   04
main:           ! Beginning of executable code for routine main
        !#PROLOGUE#     0
        save %sp, -128, %sp  ! Create 128 byte stack frame. Advance
                             ! CWP (Current Window Pointer)
        !#PROLOGUE#     1
        mov 10, %o0  ! %o0 <- 10. Note that %o0 is the same as %r24.
    ! This is local variable a in main routine of C source program.
        st %o0, [%fp-20]     ! Store %o0 five words into stack frame.
        mov 4, %o0   ! %o0 <- 4. This is local variable b in main.
        st %o0, [%fp-24]     ! Store %o0 six words into stack frame.
        ld [%fp-20], %o0     ! Load %o0 and %o1 with parameters to
        ld [%fp-24], %o1     ! be passed to routine add_two.
        call add_two, 0      ! Call routine add_two
        nop     ! Pipeline flush needed after a transfer of control
        st %o0, [%fp-28]     ! Store result 67 words into stack frame.
                             ! This is local variable c in main.
        sethi %hi(.LLC0), %o1 ! This instruction and the next load
        or %o1, %lo(.LLC0), %o0 ! the 32-bit address .LLC0 into %o0
        ld [%fp-28], %o1  ! Load %o1 with parameter to pass to printf
```

gcc Generated SPARC Code (cont')

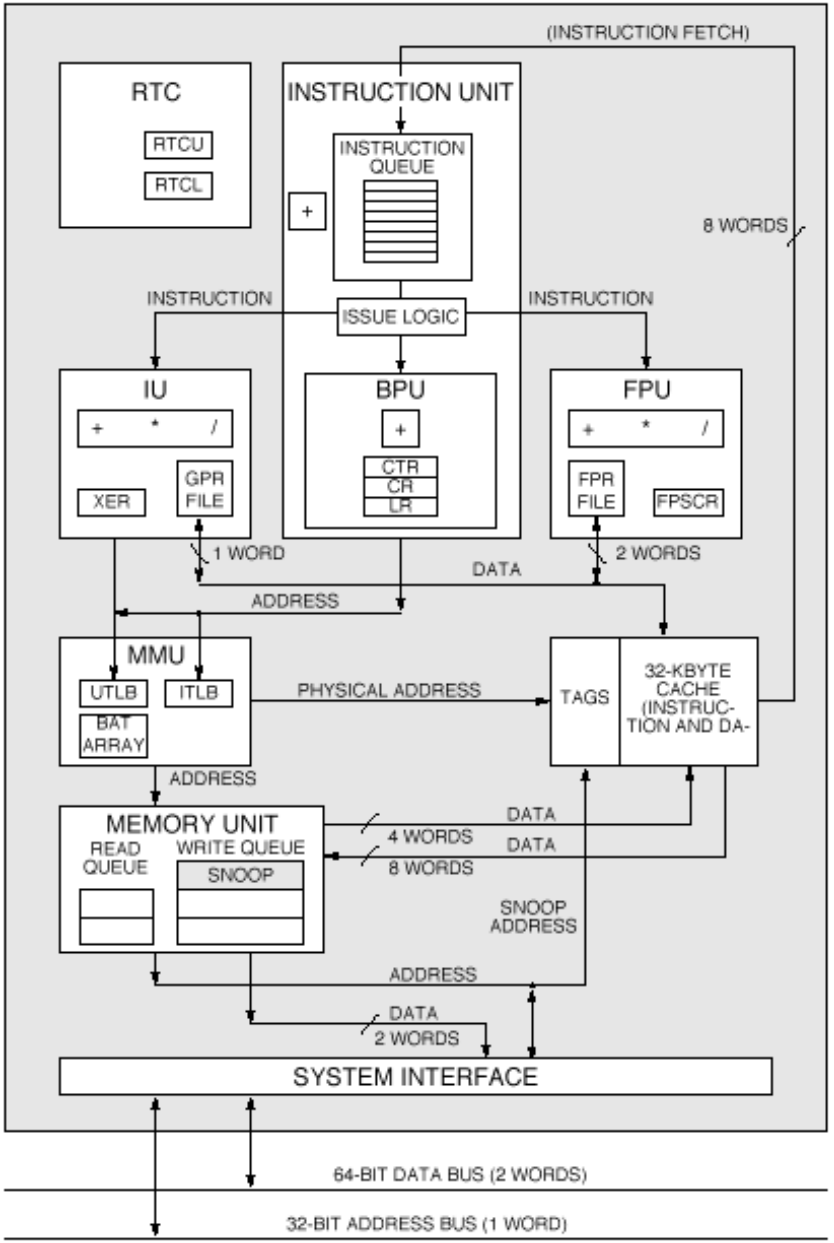
```
        call printf, 0
        nop     ! A nop is needed here because of the pipeline flush
                ! that follows a transfer of control.
.LL1
        ret     ! Return to calling routine (Solaris for this case)
        restore ! The complement to save. Although it follows the
            ! return, it is still in the pipeline and gets executed.
.LLfe1
        .size   main, .LLfe1-main ! Size of
        .align 4
        .global add_two
        .type   add_two, #function
        .proc 04
add_two:
         !#PROLOGUE# 0
         save %sp, -120, %sp
         !#PROLOGUE# 1
         st %i0, [%fp+68] !Same as %o0 in calling routine (variable a)
         st %i1, [%fp+72] !Same as %o1 in calling routine (variable b)
         ld [%fp+68], %o0
         ld [%fp+72], %o1
         add %o0, %o1, %o0 ! Perform the addition
         st %o0, [%fp-20]  ! Store result in stack frame
         ld [%fp-20], %i0  ! %i0 (result) is %o0 in called routine
         b .LL2
         nop
.LL2:
         ret
         restore
.LLfe2:
         .size   add_two, .LLfe2-add_two
         .ident  "GCC: (GNU) 2.5.8"
```

Effect of Compiler Optimization

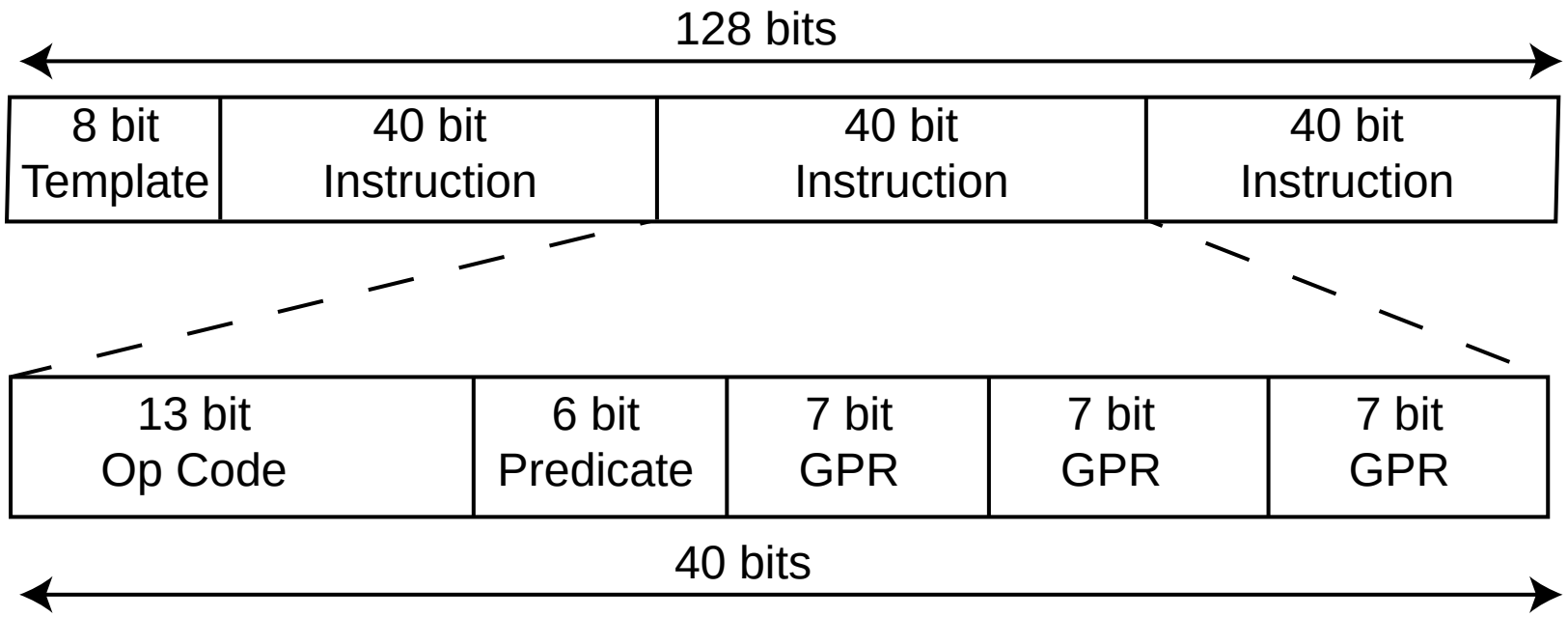
- **SPARC code generated with the -O optimization flag:**

```
! Output produced by -O optimiziation for gcc compiler

.file   "add.c"
.section        ".rodata"
        .align 8
.LLC0:
        .asciz  "c = %d\n"
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        !#PROLOGUE# 0
        save %sp,-112,%sp
        !#PROLOGUE# 1
        mov 10,%o0
        call add_two,0
        mov 4,%o1
        mov %o0,%o1
        sethi %hi(.LLC0),%o0
        call printf,0
        or %o0,%lo(.LLC0),%o0
        ret
        restore
.LLfe1:
        .size    main,.LLfe1-main
        .align 4
        .global add_two
        .type    add_two,#function
        .proc   04
add_two:
        !#PROLOGUE# 0
        !#PROLOGUE# 1
        retl
        add %o0,%o1,%o0
.LLfe2:
        .size    add_two,.LLfe2-add_two
        .ident  "GCC: (GNU) 2.7.2"
```

The PowerPC 601 Architec-ture



128-Bit IA-64 Instruction Word



Parallel Speedup and Amdahl's Law

- **In the context of parallel processing, speedup can be computed:**

$$S = \frac{T_{Sequential}}{T_{Parallel}}$$

- **Amdahl's law, for *p* processors and a fraction *f* of unparallelizable code:**

$$S = \frac{1}{f + \frac{1-f}{p}}$$

- **For example, if *f* = 10% of the operations must be performed sequentially, then speedup can be no greater than 10 regardless of how many processors are used:**

$$S = \frac{1}{0.1 + \frac{0.9}{10}} \cong 5.3 \qquad S = \frac{1}{0.1 + \frac{0.9}{\infty}} = 10$$

$$p = 10 \text{ processors} \qquad p = \infty \text{ processors}$$

Efficiency and Throughput

- ***Efficiency*** **is the ratio of speedup to the number of processors used. For a speedup of 5.3 with 10 processors, the efficiency is:**

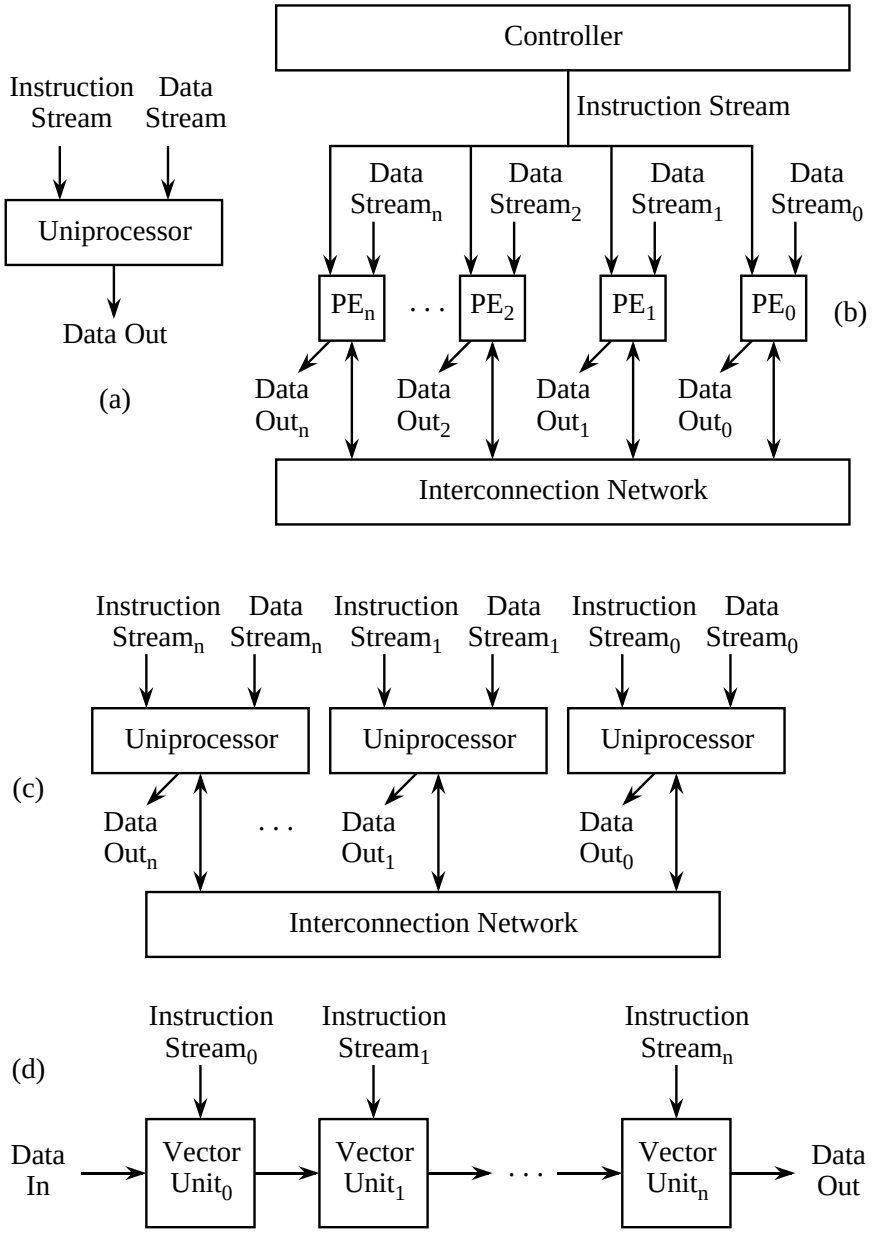
$$\frac{5.3}{10} = .53, \text{ or } 53\%$$

- ***Throughput*** **is a measure of how much computation is achieved over time, and is of special concern for I/O bound and pipelined applications. For the case of a four stage pipeline that remains filled, in which each pipeline stage completes its task in 10 ns, the average time to complete an operation is 10 ns even though it takes 40 ns to execute any one operation. The overall throughput for this situation is then:**

$$0.1 \frac{operation}{ns} = 10^8 \text{ operations per second.}$$

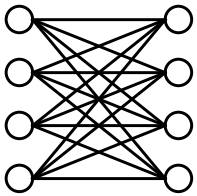
Flynn Taxonomy

- **Classification of architectures according to the Flynn taxonomy: (a) SISD; (b) SIMD; (c) MIMD; (d) MISD.**

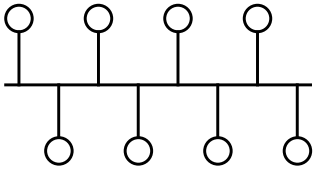


Network Topologies

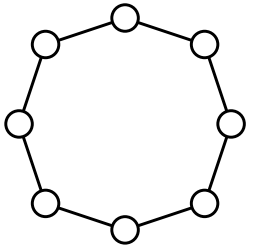
- **Network topologies: (a) crossbar; (b) bus; (c) ring; (d) mesh; (e) star; (f) tree; (g) perfect shuffle; (h) hypercube.**



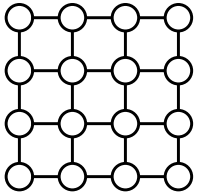
(a)



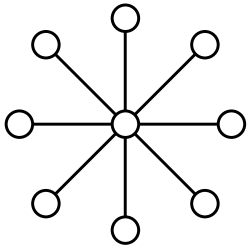
(b)



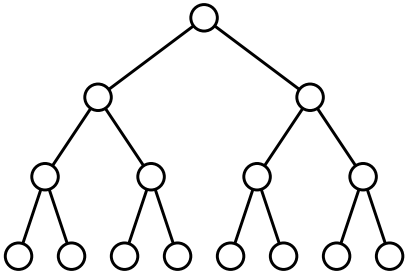
(c)



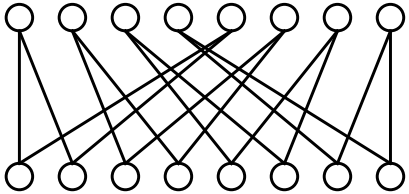
(d)



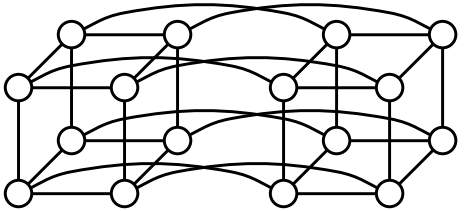
(e)



(f)



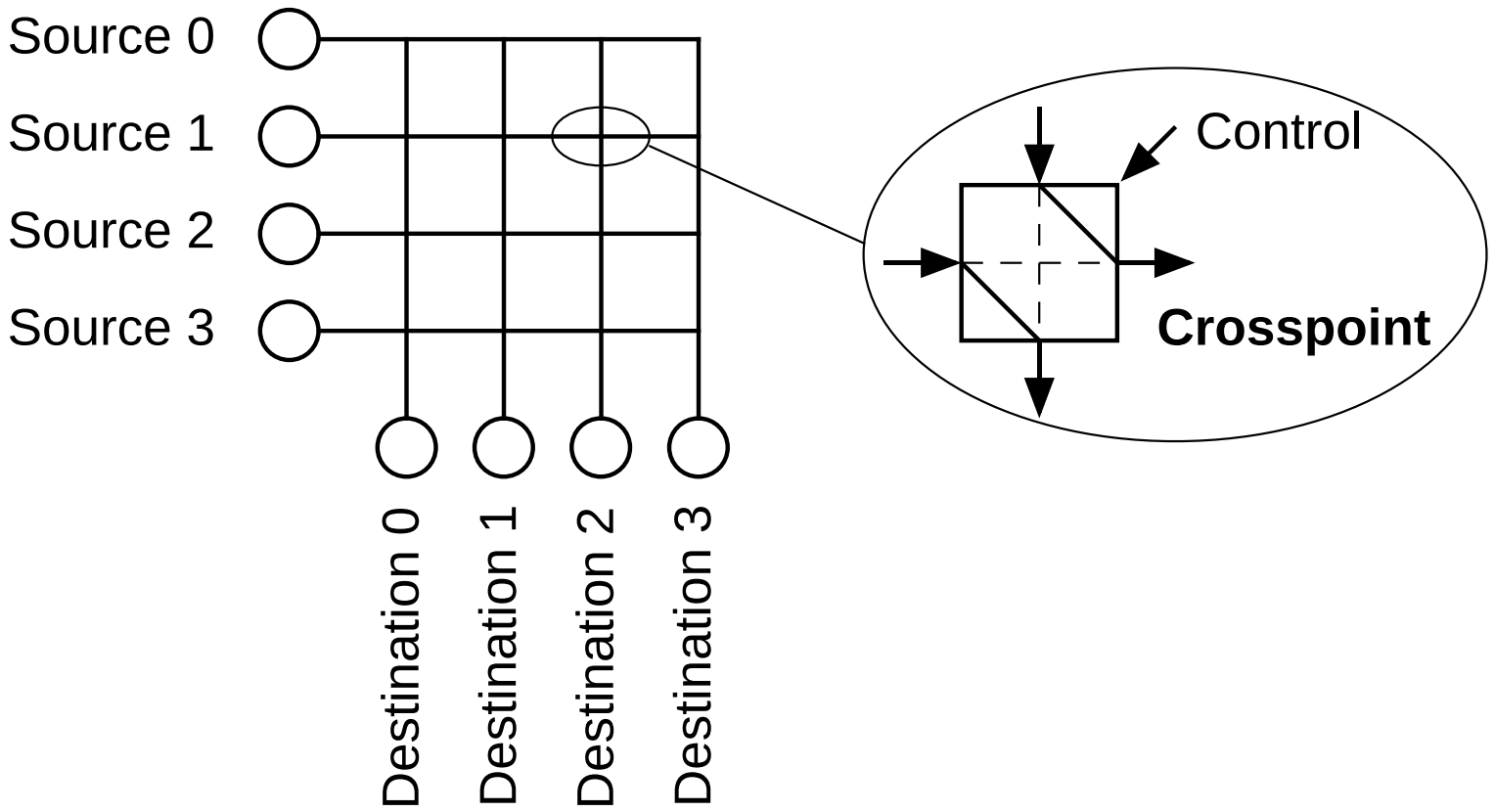
(g)



(h)

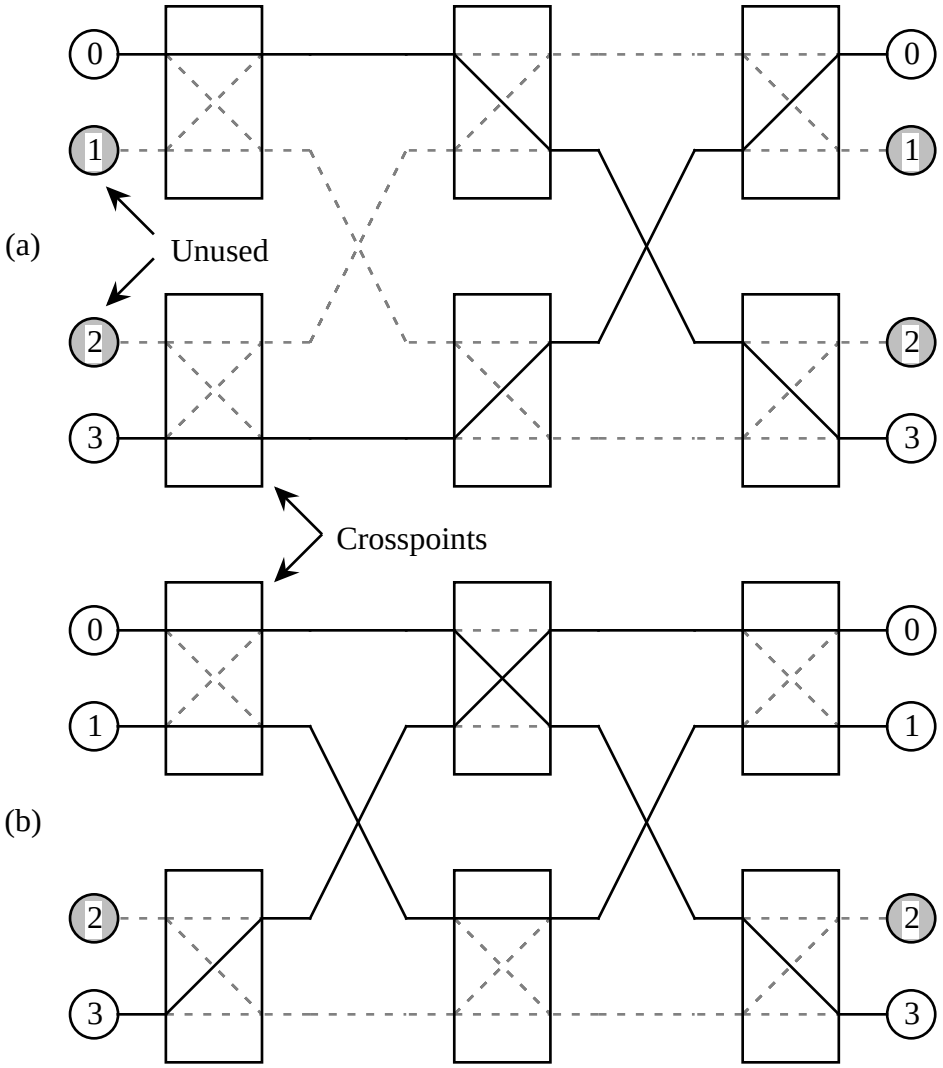
Crossbar

- **Internal organization of a crossbar.**

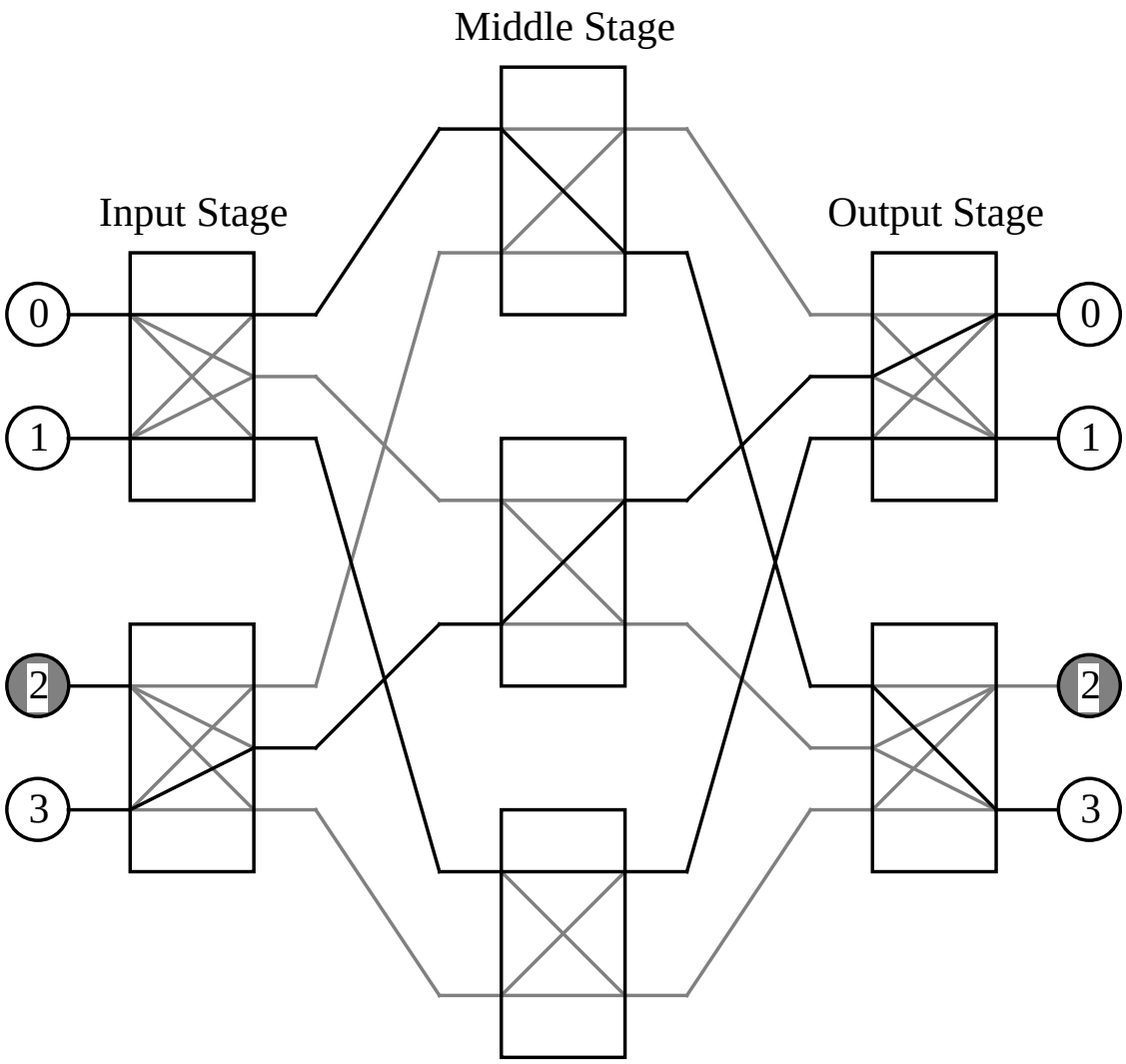


Crosspoint Settings

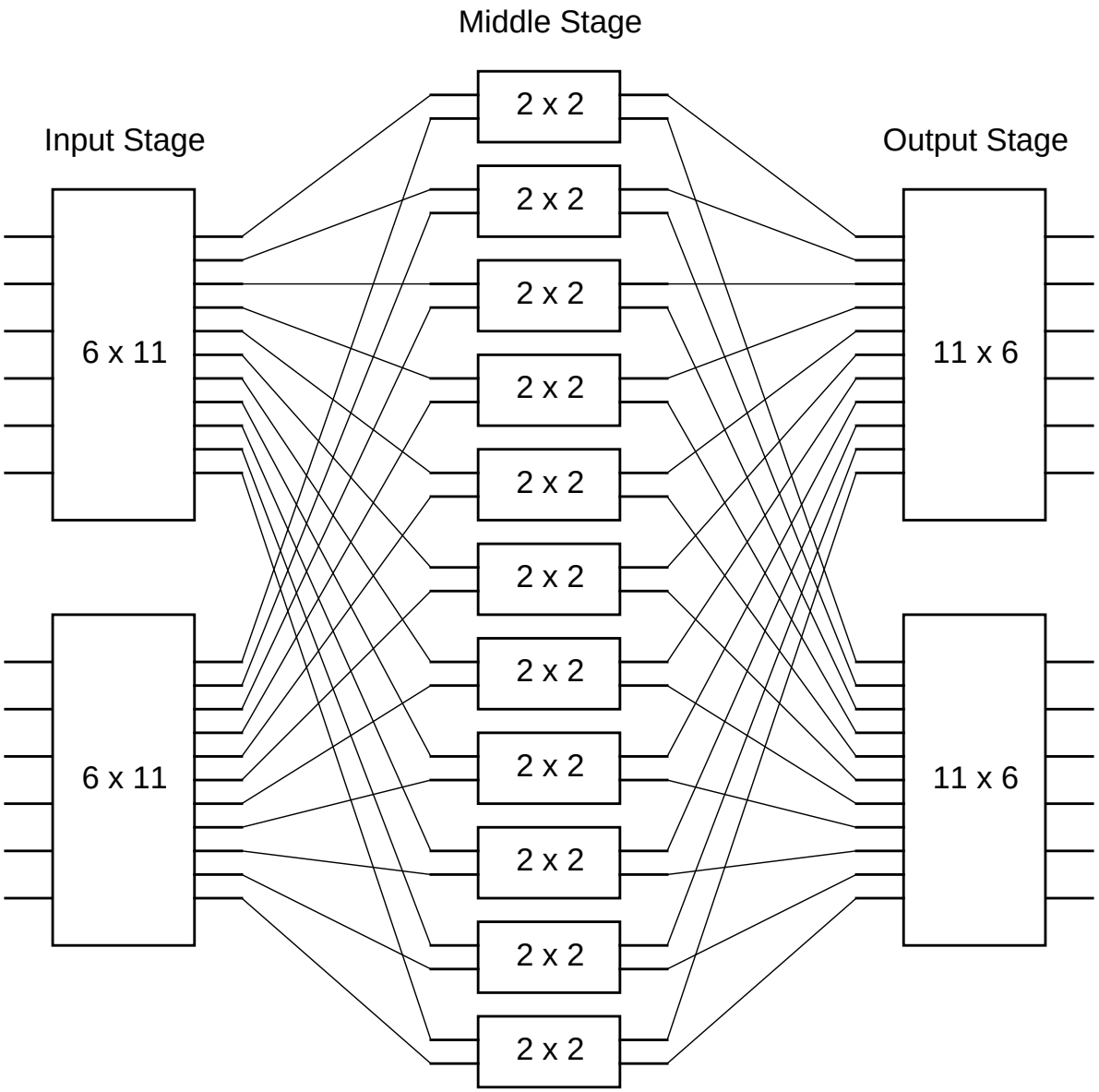
- **(a) Crosspoint settings for connections 0 → 3 and 3 → 0; (b) adjusted settings to accommodate connection 1 → 1.**



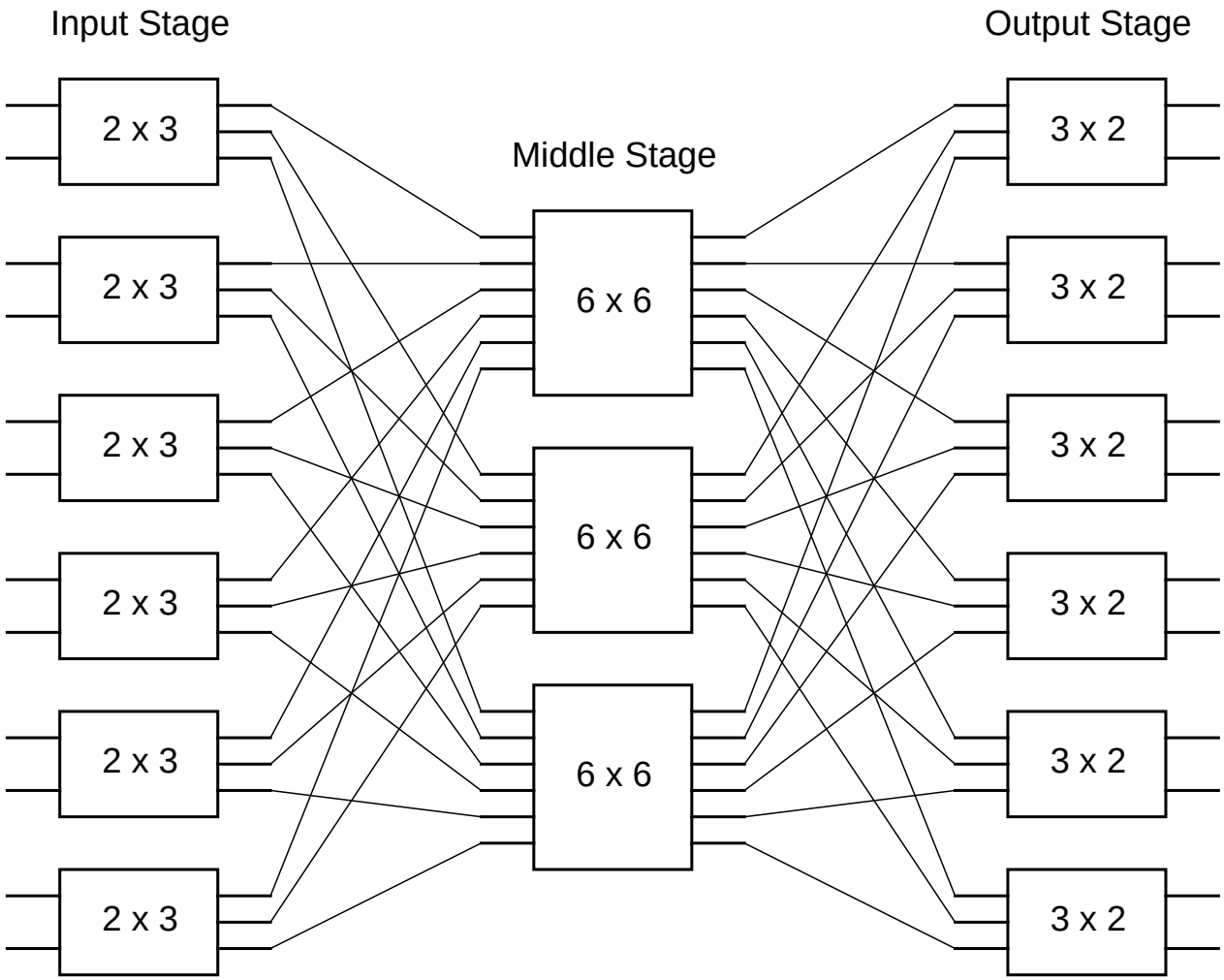
Three-Stage Clos Network



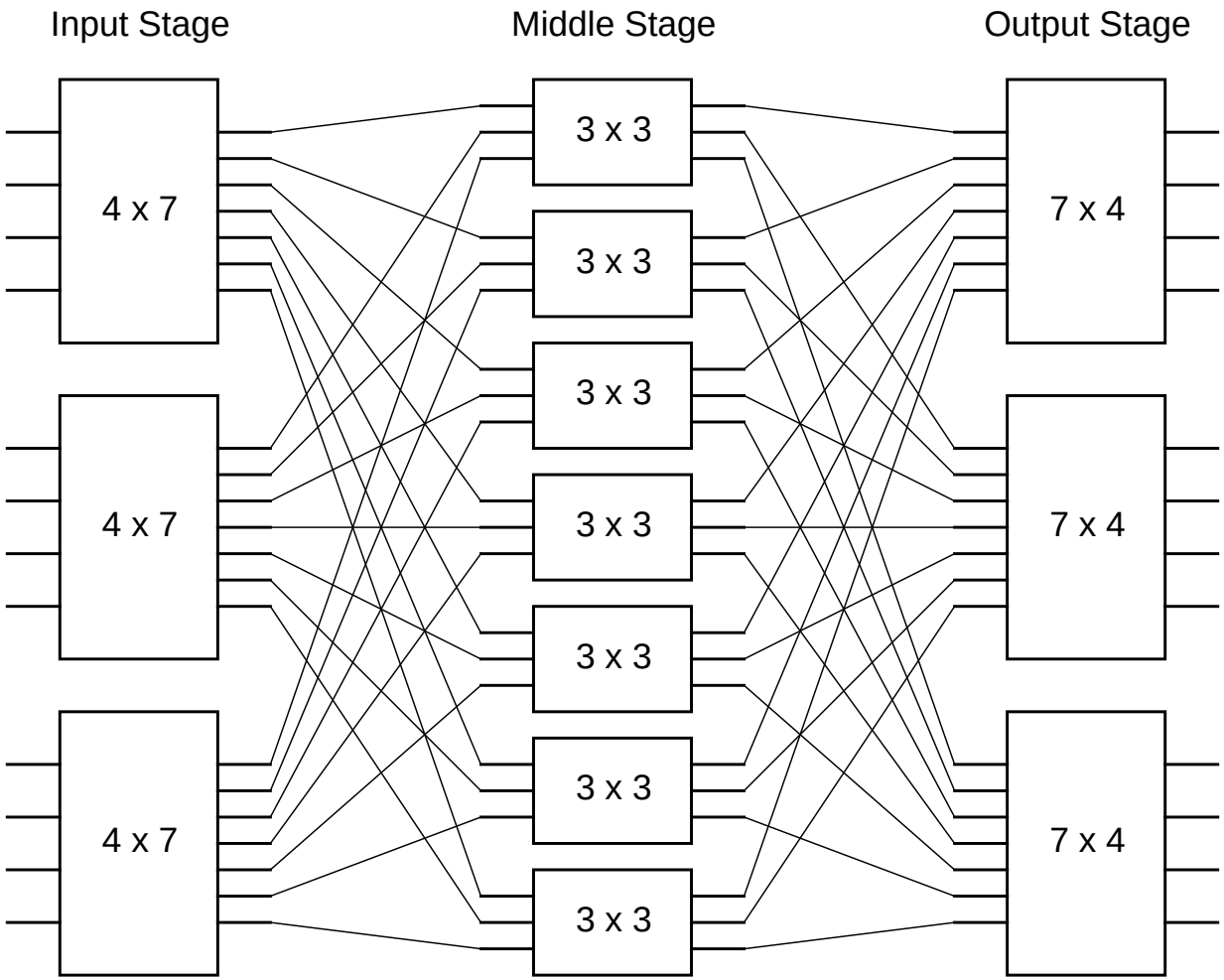
12-Channel Three-Stage Clos Network with $n = p = 6$



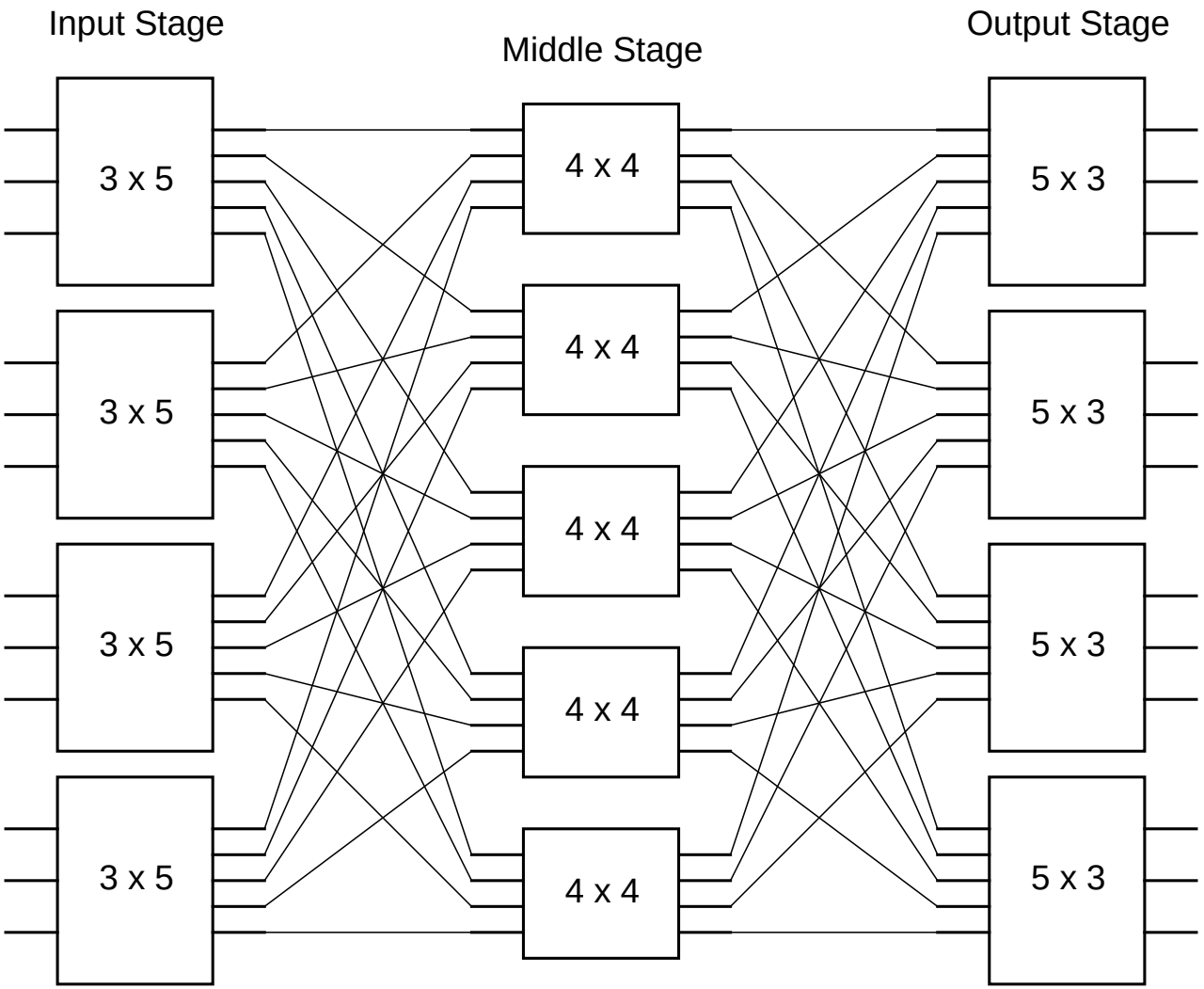
12-Channel Three-Stage Clos Network with $n = p = 2$



12-Channel Three-Stage Clos Network with $n = p = 4$



12-Channel Three-Stage Clos Network with $n = p = 3$



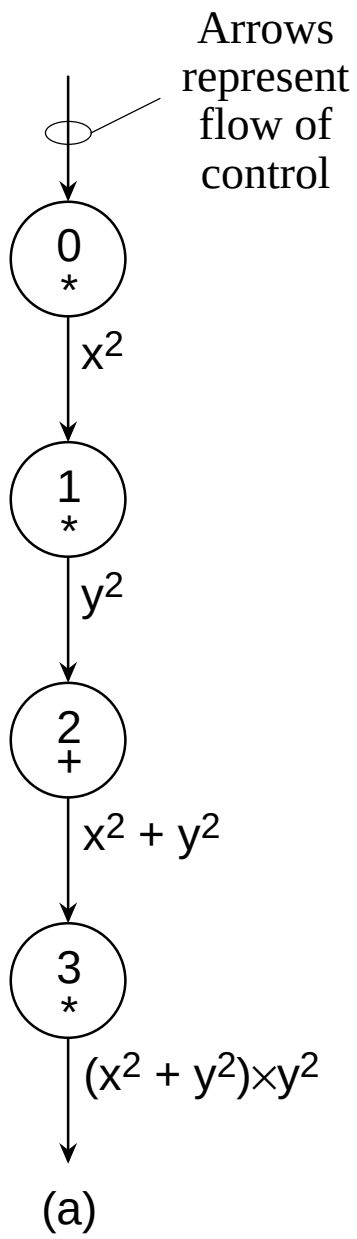
C function computes $(x^2 + y^2) \times y^2$

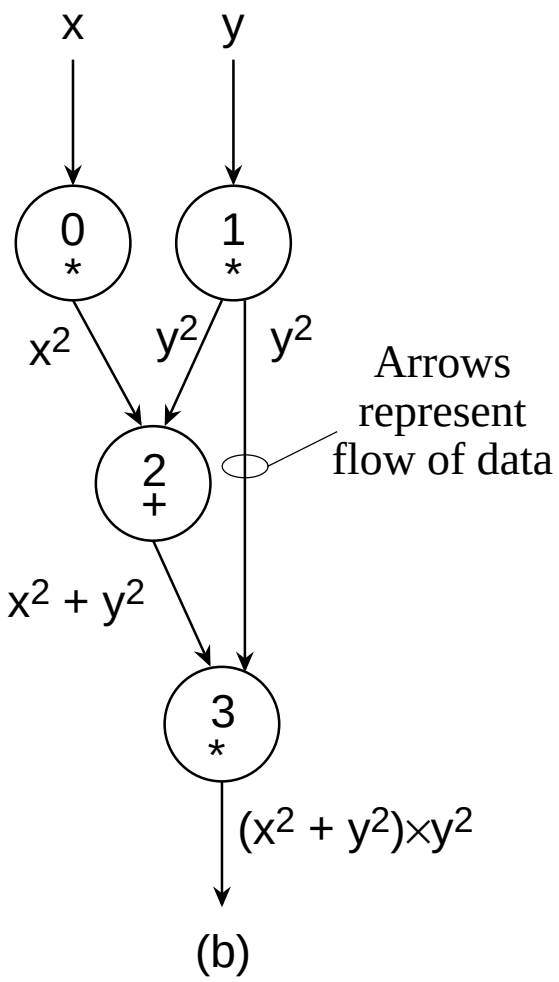
Operation numbers

```
        func(x, y)  /* Compute (x² + y²) × y² */
        int x, y;
        {
        int temp0, temp1, temp2, temp3;

0       temp0 = x * x;
1       temp1 = y * y;
2       temp2 = temp1 + temp2;
3       temp3 = temp1 * temp2;

        return(temp3);
        }
```

Dependency Graph

- **(a) Control sequence for C program; (b) dependency graph for C program.**





Matrix Multiplication

- **(a) Problem setup for Ax = b; (b) equations for computing the b_i.**

(a)

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

(b)

$$\overset{0}{\overset{4}{b_0} = a_{00}x_0} + \overset{1}{\overset{6}{a_{01}x_1}} + \overset{2}{\overset{5}{a_{02}x_2}} + \overset{3}{a_{03}x_3}$$

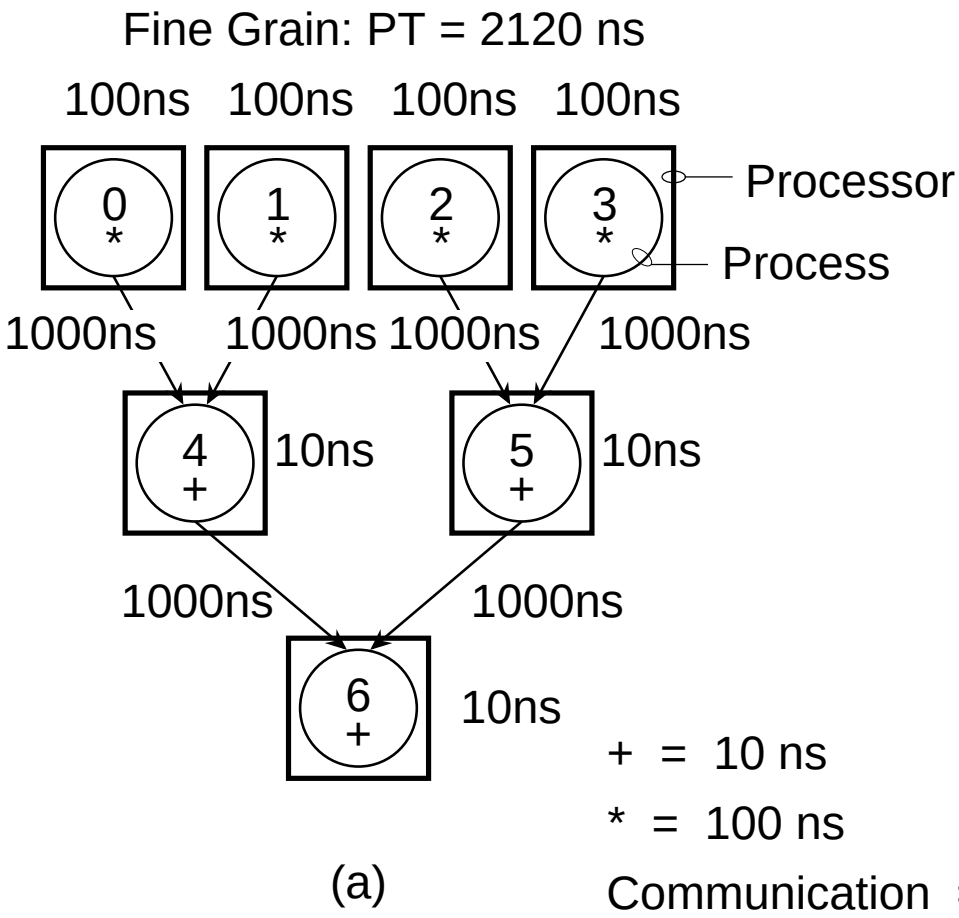
$$\overset{7}{\overset{11}{b_1} = a_{10}x_0} + \overset{8}{\overset{13}{a_{11}x_1}} + \overset{9}{\overset{12}{a_{12}x_2}} + \overset{10}{a_{13}x_3}$$

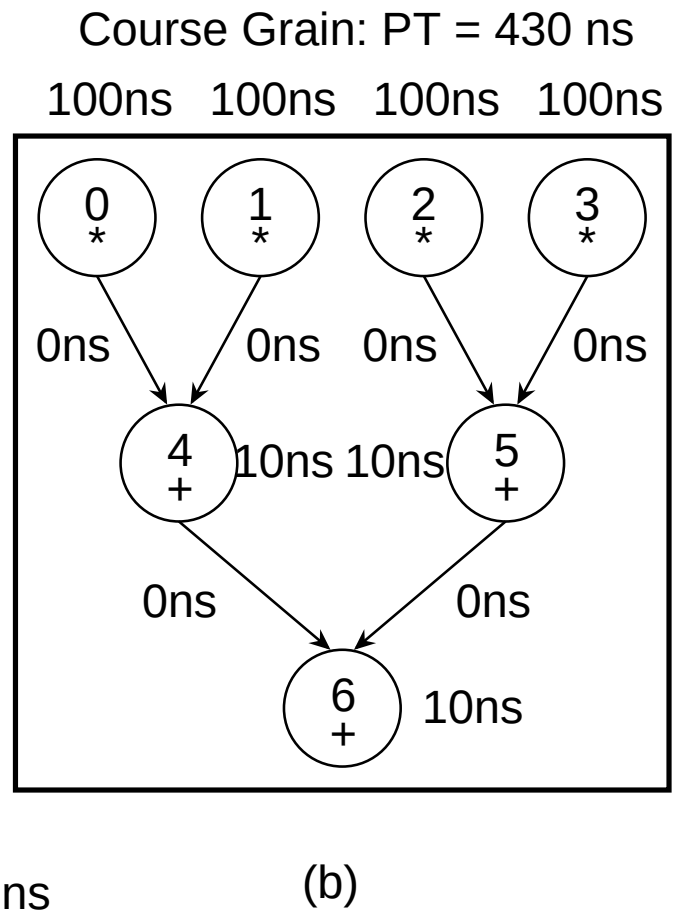
$$\overset{14}{\overset{18}{b_2} = a_{20}x_0} + \overset{15}{\overset{20}{a_{21}x_1}} + \overset{16}{\overset{19}{a_{22}x_2}} + \overset{17}{a_{23}x_3}$$

$$\overset{21}{\overset{25}{b_3} = a_{30}x_0} + \overset{22}{\overset{27}{a_{31}x_1}} + \overset{23}{\overset{26}{a_{32}x_2}} + \overset{24}{a_{33}x_3}$$

Matrix Multiplication Dependency Graph

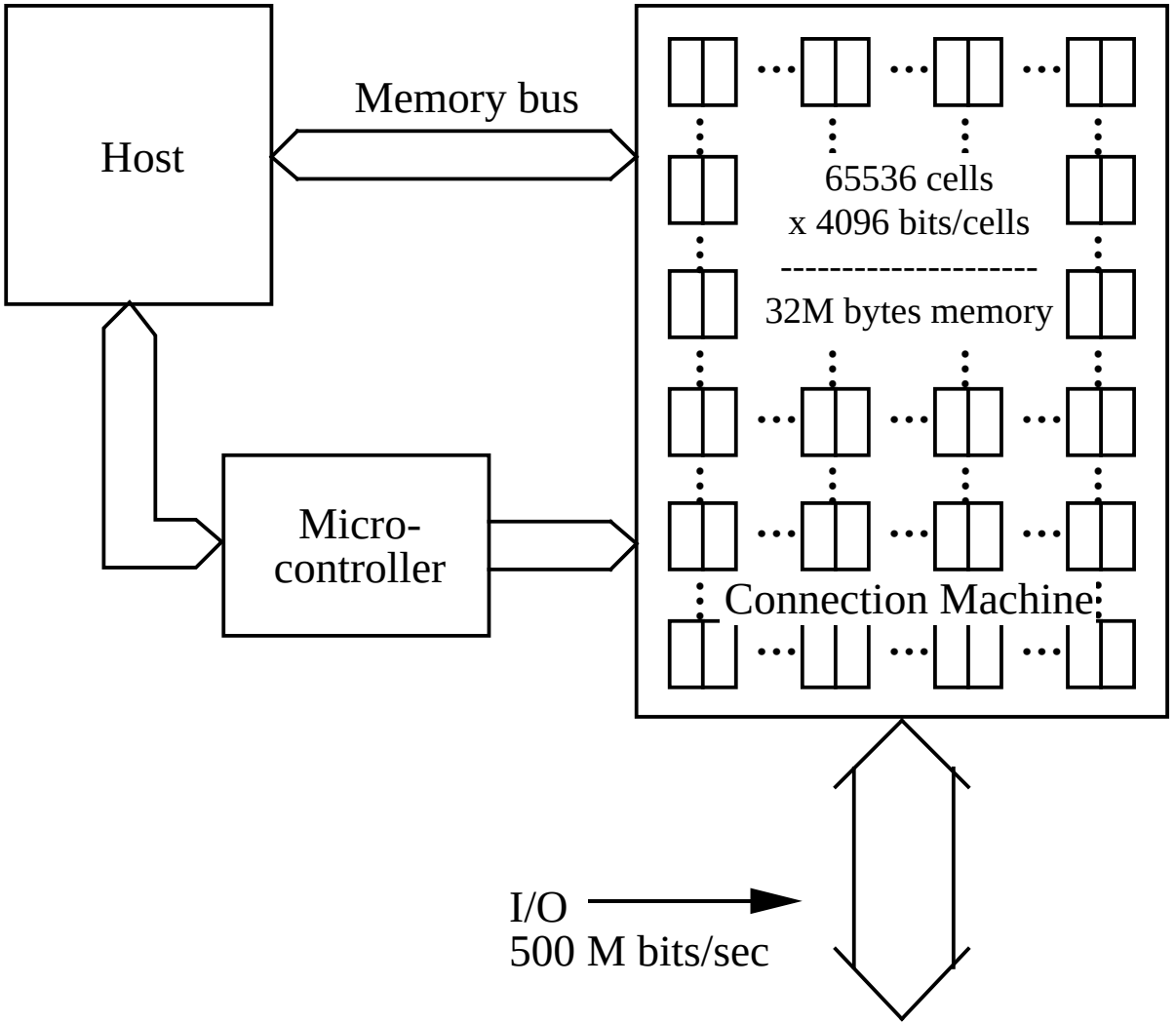
$$\frac{T_{Sequential}}{T_{Parallel}} = \frac{1720}{430} = 4$$





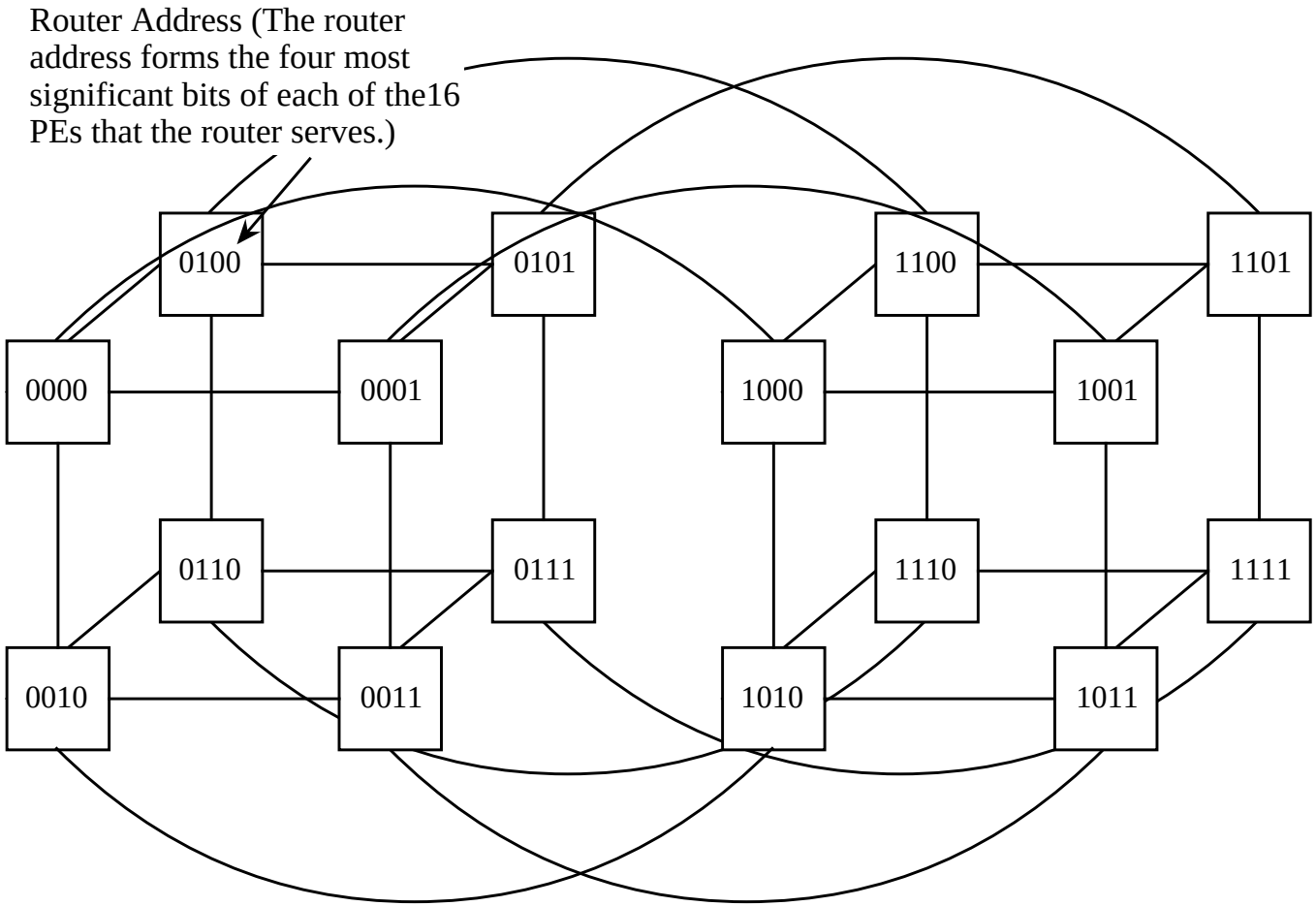
The Connection Machine CM-1

- **Block diagram of the CM-1 (Adapted from Hillis, W. D., *The Connection Machine*, The MIT Press, 1985).**

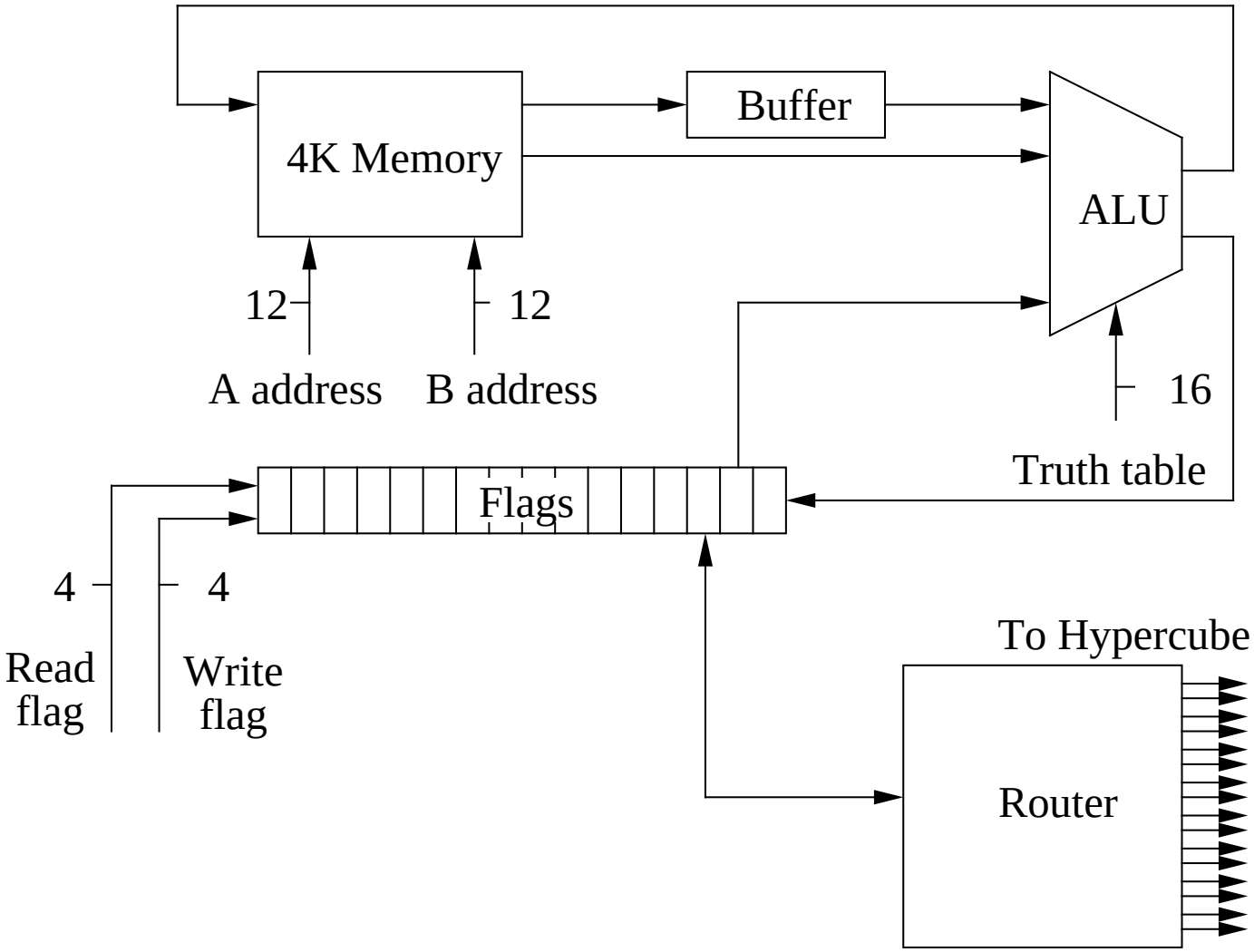


CM-1 Router Network

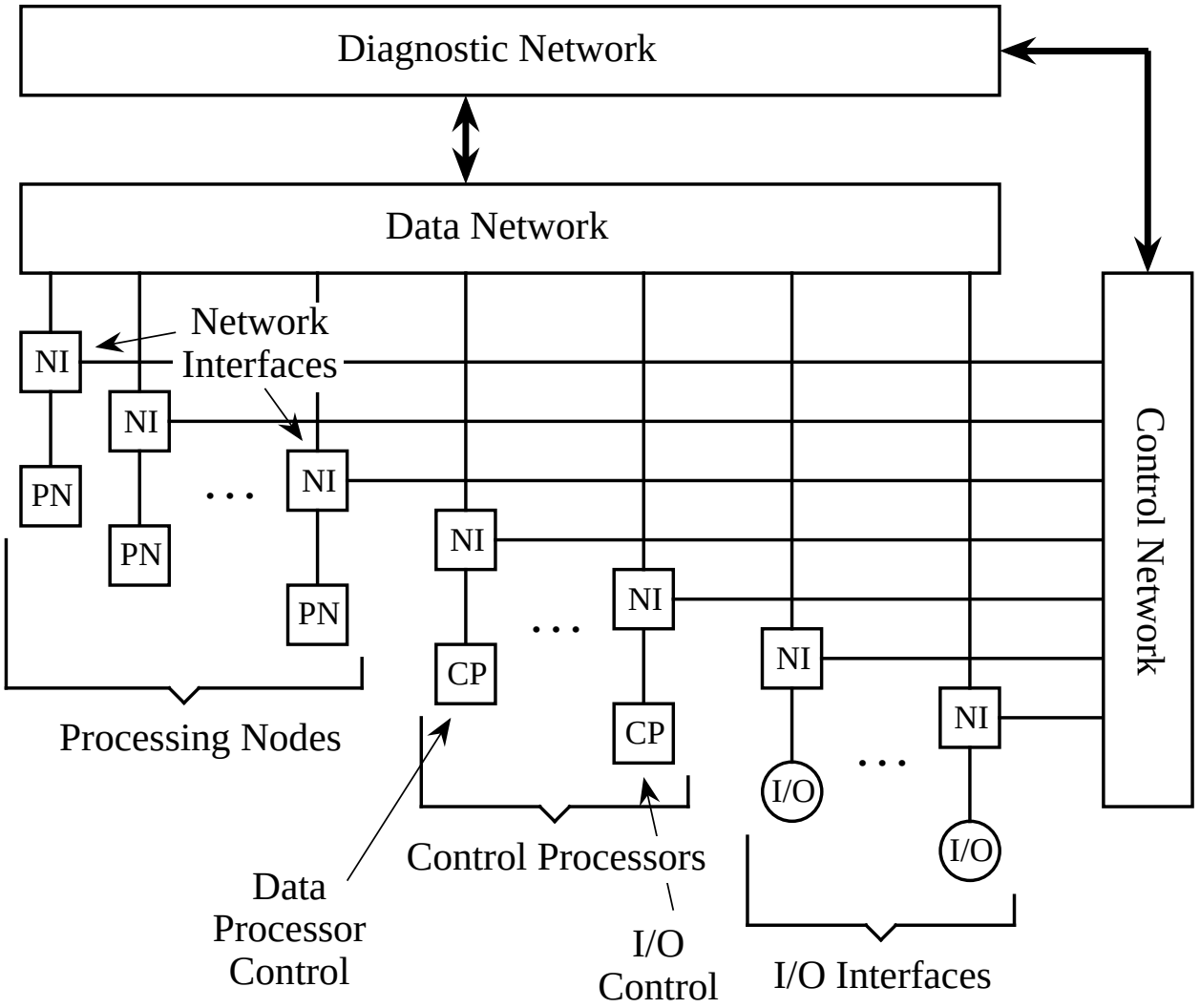
- **A four-space hypercube for the router network.**



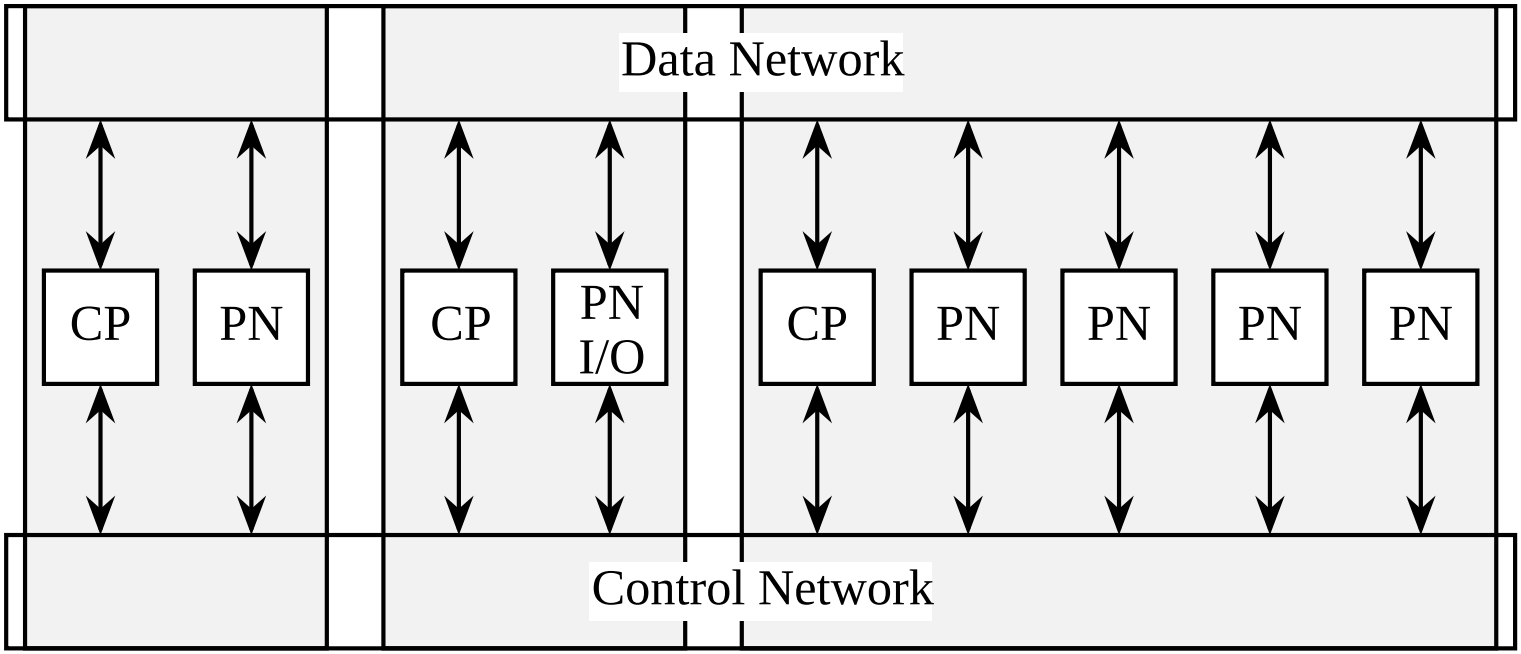
CM-1 Processing Element



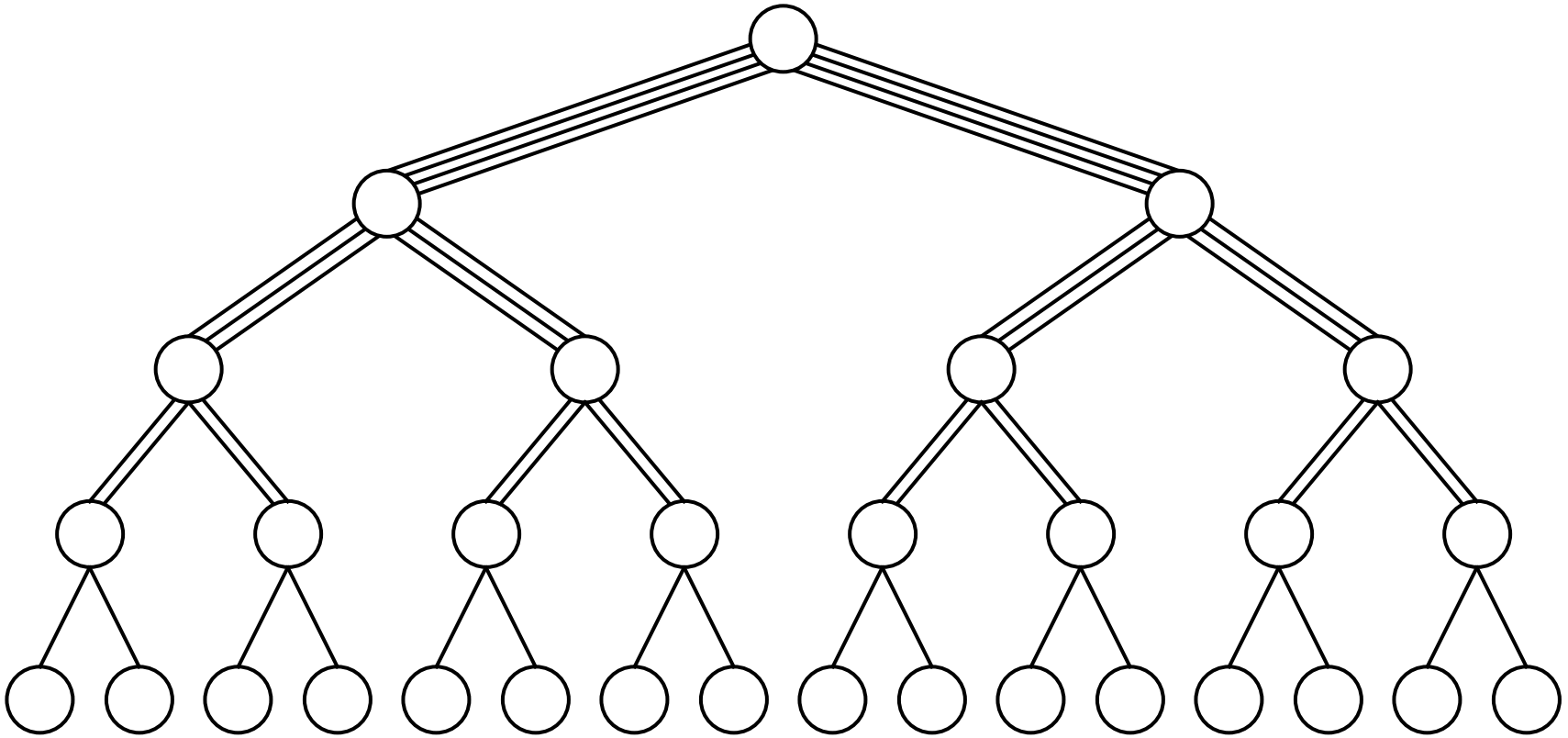
The Connection Machine CM-5



Partitions on the CM-5

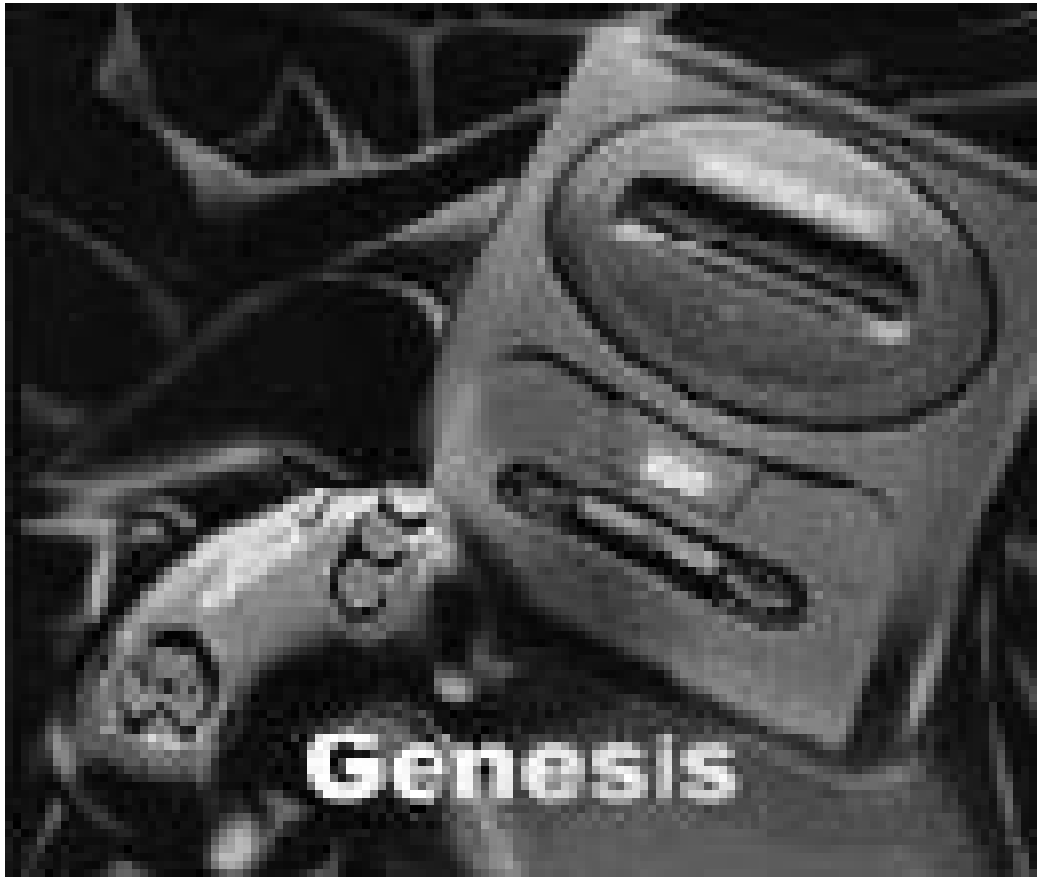


Fat Tree



Parallel Processing in Sega Genesis

- **External view of the Sega Genesis home video game system.**



Sega Genesis Architecture

- **System bus model view of the Sega Genesis.**