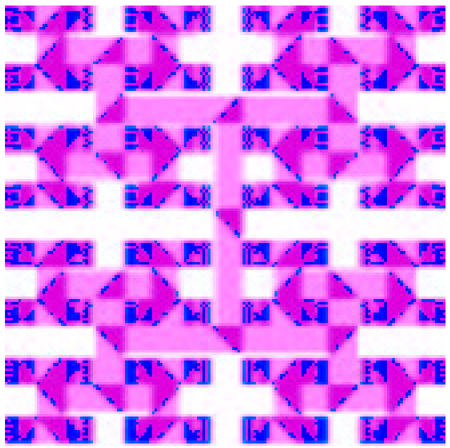

Ultrascalar Memo 2

An Efficient, Prioritized Scheduler Using Cyclic Prefix

Dana S. Henry and Bradley C. Kuszmaul

November 23, 1998

Using Cyclic Segmented Parallel Prefix (CSPP) circuits [1], we show in this memo how to design scheduler circuits that efficiently assign a set of resources to a set of requesting elements. The elements are assumed to be stored in a wrap-around sequence. Not all the elements in the sequence need be requesting and the number of requesting elements may exceed the number of available resources. Various criteria can be used to select which requesting elements receive the resources. For example, one of the described circuits assigns the resources to the oldest requesting elements. A more elaborate circuit subdivides elements into several priority levels and assigns resources to higher priority level elements first and, within each level, to older elements first. Additional circuitry enables the assignment of only a subset of resources.

These scheduler circuits can be used to implement the scheduling stage of a superscalar processor which selects a set of ready instructions to run on functional units. The instructions are in reservation stations arrayed in the instruction window. Each reservation station contains information indicating whether its instruction is ready to run, and which functional unit is needed. The functional units include floating point adders, multipliers, integer units, and memory units. At the beginning of every clock cycle, some of the instructions in the reservation stations are ready to run. There may be more than one functional unit that can be used by any particular instruction. In many situations, certain of the instructions should be scheduled with higher priority than others. For example, it may be prefered to schedule the oldest instructions with higher priority, or it may be preferred to schedule instructions which resolve condition codes with higher priority. For memory operations it may be preferable to schedule nonspeculative loads rather than speculative loads.

We divide the functional units into classes. For our example, we have the adders be Class 0, the dividers be Class 1, and the memory units be Class 2. Let us assume there are three adders, two dividers, and one memory unit. The number of functional units in Class i is denoted by F_i. The functional units in Class i are numbered 0 to $F_i - 1$. All F_i functional units are available on every clock cycle.

After scheduling the instructions, the data must actually be moved to the appropriate functional units.

Scheduling The Instructions

Figure 1 shows the logic that assigns functional units to ready instructions. From the analyze dependencies stage come many "request" bits labeled $Q_{i,j}$. Bit $Q_{i,j}$ is true if and only if ith instruction in the window is ready to run and it needs a functional unit from Class j. To avoid cluttering up the figure, the signals $Q_{1,j}$ are collectively labeled $Q_{1,*}$, and the signals $Q_{2,j}$ are collectively labeled $Q_{2,*}$ and the last window entry's request bits are labeled $Q_{W-1,*}$. The logic produces signals $G_{i,j}$ (with a similar notation that $G_{i,*}$ refers to all the request signals from Window i) which go back to the reordering buffer in the analysis stage. Signal $G_{i,j}$ may require several bits to encode, since it must encode at least $1 + F_i$ different values (one for each functional unit of that class, and one value to indicate that no functional unit was assigned.) Hence $G_{i,j}$ must be at least $\lceil \lg(1 + F_i) \rceil$ bits wide. Several different encodings are possible. For concreteness, we will use the following encoding for $G_{i,j}$:

$$G_{i,j} = \begin{cases} k & \text{if } i\text{the instruction in the window is assigned to function unit Number } k \text{ in Class } j, \\ F_i & \text{if } i\text{the instruction in the window requested a function unit and the request was denied,} \\ \text{don't care} & \text{if } i\text{the instruction in the window did not request a function unit.} \end{cases}$$

In Figure 1 there are three boxes labeled "schedule adder", "schedule divider", and "schedule memory unit", respectively. These boxes are the schedulers for each of our three example functional unit classes. Each of these is simply a CSPP summation circuit [1], summing 1-bit inputs $Q_{i,j}$, followed by W comparators that compute $\min(y_j, F_i)$.

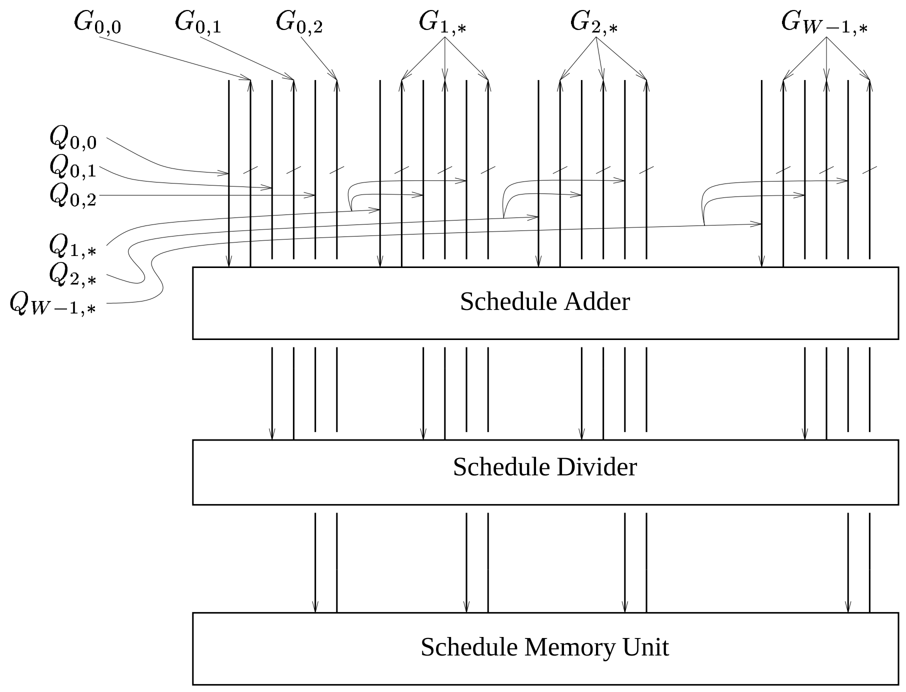

Figure 1: Schedule Assignment

Specifically, for Class i we have

$$\begin{aligned} s_j &= \text{the bit indicating that Window Entry } j \text{ is oldest.} \\ x_j &= Q_{i,j} \\ G_{i,j} &= \min(y_j, F_i) \end{aligned}$$

Thus if $y_j < F_i$ then $G_{i,j}$ indicates that Window Entry j receives functional unit Number y_j in Class j.

Note that there are some special cases that can be optimized to reduce the circuit complexity. Typically there are more window entries than functional units in a class. To compute $\lceil \lg W \rceil$ bits and then take the minimum of that number with F_i is wasteful. Instead of doing that, a saturating adder can be used in the tally circuit so that the number of bits going up and down the tree never becomes larger than $\lceil \lg(1 + F_i) \rceil$ bits in size.

An important special case is if $F_i = 1$, (for example, we only have one memory unit in our example.) In this case an OR gate implements a saturating adder, and the circuit becomes the one shown in Figure 2.

Moving The Data

To move the data to the functional units, several different strategies can be used, including, for example, a broadcast bus or a tree of busses, or a broadcast tree. For very large issue widths and very large numbers of functional units, it is important that the data movement mechanism scale well. (That is, large amounts of data need to move with low latency.) Here we show how to use a butterfly routing network to implement the data movement. This butterfly has enough bandwidth to move all the data in all cases without introducing contention at the internal switches, and it can move the data with only a logarithmic number of gate delays.

Figure 3 shows a butterfly network implemented with switches of degree two. (See, for example, [2, 4] for methods for routing in this kind of network.) This network can move data to the three adders from eight window locations. The switches are connected in a butterfly network. The addresses are the $G_{i,j}$ values provided by the scheduler CSPP circuits. The data are the two register values that the reorder buffer provides. Shown is a signal containing the address and a signal containing data from each of the 8 reorder buffer slots.

Figure 4 shows an example of how this works. We have deleted a few of the switches which were not needed since we had fewer destinations than sources. The thick gray line shows the path followed to get data from Sources 1, 2, and 7 to Destinations 0, 1, and 2 respectively. The path chosen is the one chosen by the algorithm of [2]. Specifically, to

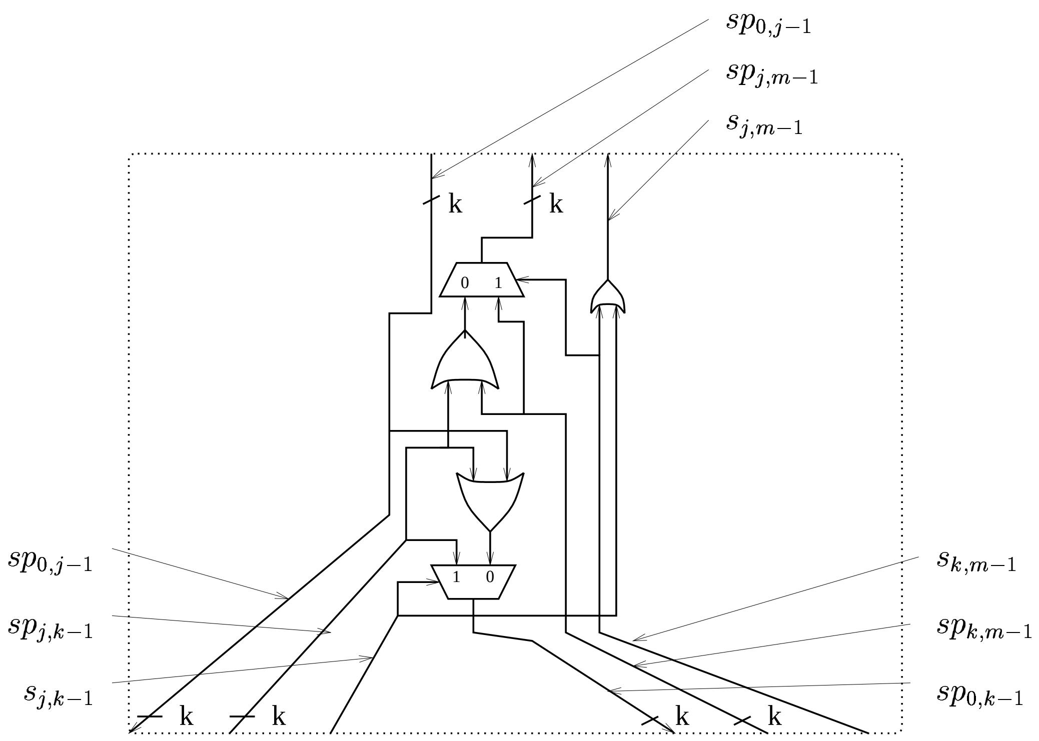

Figure 2: A disjunction module implements a 1-bit saturating adder.

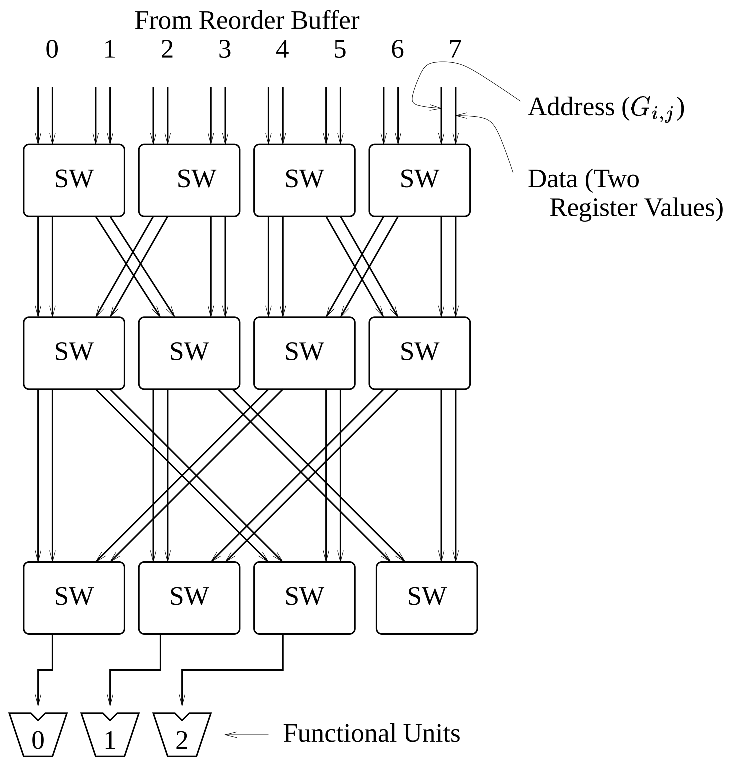

Figure 3: The Data Movement Network

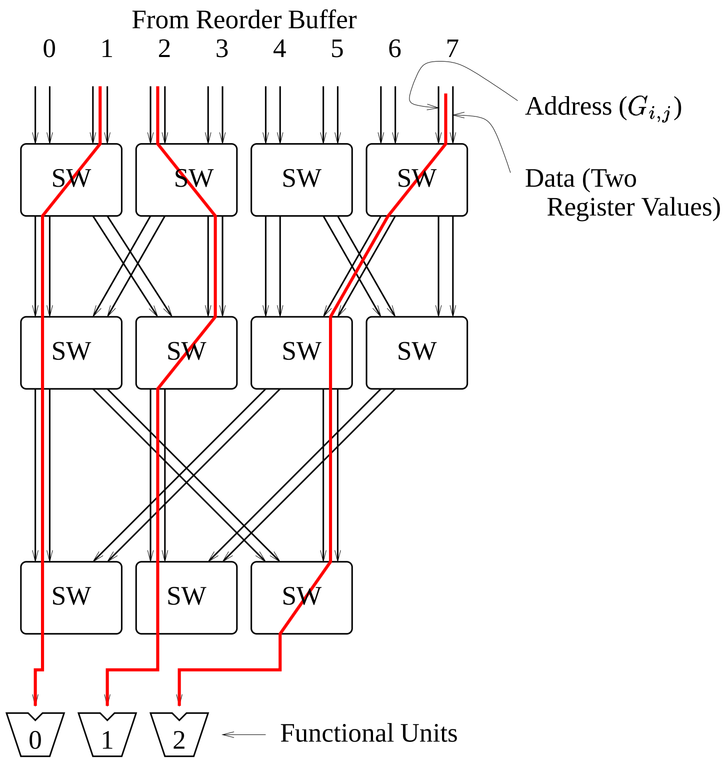

Figure 4: Moving Data

get from Source i to Destination j one computes the *relative address* $\mathrm{XOR}(i, j)$, which is the bitwise exclusive or of i and j. Then the message goes straight through switch k if bit k is false in the relative address. Thus to go from Source 1 to Destination 0, the relative address is 1 which is 001_2 in base 2. The message should go diagonally through the first switch and straight through the next two switches. To go from Source 7 to Destination 2, the relative address is $101_2 = \mathrm{XOR}(111_2, 010_2)$, so the message should go diagonally through the first and third switch, and straight through the second switch. Note that [2] uses a wormhole router, but other routing mechanisms can be used to establish the path through the circuit, since the circuit is all one one VLSI chip.

Note that for the example we showed there is no conflict for the use of the wires. It turns out that for all the routing patterns produced by this scheduling circuit, there are no conflicts, and the messages can go through in one clock cycle. This is because our scheduler produces only routing patterns that are *semi-contractions*. A semi-contraction is a routing which has the property that the distance between the destinations of two messages is no greater than the distance between their initial locations. I.e., $|i - j| \geq |f(i) - f(j)|$. In [3] a proof is given that semi-contractions can be routed on a butterfly with no conflicts.

Many of the other networks, such as fat trees, omega networks, complete networks, busses, or trees of busses can be used. Engineering considerations should include taking into account the exact size of the window, the number of functional units, and the number of function unit classes. For example, for a small system a bus may be the best solution, whereas for a large system, a butterfly of Radix 8 may be the best solution. For any particular problem size, determining the costs of the various options is straightforward, however. Also, Several different VLSI layouts are possible, including the layout shown in Figure 3 or an H-tree layout. See [4] for a complete discussion of networks for parallel computing.

Scheduling Certain Instructions with Higher Priority

Usually a scheduler that prefers older instructions rather than younger instructions is a good heuristic. Sometimes, certain younger instructions are preferred over older ones. For example, instructions that resolve conditional branches are good to execute. To schedule such instructions with higher priority we must first identify the instructions, and then schedule the identified instructions.

It is straightforward to use CSPP to identify an instruction that resolves a condition code for an unresolved branch.

One performs a backwards CSPP with

$$\begin{aligned} \otimes &= \text{“and”}, \\ s_i &= \text{true for unresolved branches, and} \\ x_i &= \text{true for instructions that do not resolve condition codes.} \end{aligned}$$

In this case y_i says that there are no instructions that resolve condition codes until the next unresolved branch. If an instruction resolves condition codes and y_i is true (i.e., $y_i \wedge \overline{x_i}$), then Instruction i is identified as a higher-priority instruction.

Among the higher priority instructions, one can schedule the oldest one using a CSPP scheduler as decribed above. After scheduling the high priority instructions, one uses another CSPP circuit to schedule the low priority instructions. To account for the fact that some of the functional units were allocated to higher priority instructions, one simply adds the number of previously allocated functional units to the output of the tally circuit to compute the assignment of instructions to functional units in the ordinary-priority instructions. Note that the resulting routing of data may not be conflict-free through the hypercube, however. There are several ways to solve this problem. One way is to design a network that can handle the resulting routings. One such network consists of two butterflies, one for the higher priority instructions and one for the lower priority instructions. Another approach is to take advantage of the fact that once we have decided which instructions are to be allocated to a function unit, we do not really care which function unit in the class is allocated to which instruction. One can take advantage of this leeway by running an additional parallel prefix operation: after identifying all the instructions that will be assigned a functional unit, one then tallys those instructions. An acyclic unsegmented parallel prefix will do the job in this case with

$$\begin{aligned} x_i &= \text{true iff Instruction } i\text{'s first assignment was less than } F_j\text{, and} \\ y_i &= \text{the actual functional unit to which Instruction } i \text{ is assigned.} \end{aligned}$$

Then one can route the allocated nstructions to their functional units through a butterfly switch with no conflicts.

Scheduling a Subset of Functional Units

Note that so far we have assumed that all F_i functional units can scheduled on every clock cycles. This is true of pipelined functional units such as a pipelined multiplier or 1-cycle functional units such as an ALU. However, it is not true of multi-cycle functional units such as a divider. Consider the scenario where a processor has eight dividers that comprise Class 0. Dividers 3 and 7 are available to be scheduled. We wish to match dividers 3 and 7 with the two oldest requesting instructions, i and j. The two oldest requesting instructions received $G_{i,0} = 0$ and $G_{j,0} = 1$ respectively. To inform instructions i and j that they have been assigned dividers 3 and 7 respectively, we first use a summation parallel prefix to number available dividers:

$$x_j = \text{1 if divider is available, 0 otherwise.}$$

We then connect the scheduled instructions to available dividers using two back to back butterfly networks. The routing is collision free.

References

[1] Dana S. Henry and Bradley C. Kuszmaul. Cyclic segmented parallel prefix. Ultrascalar Memo 1, Yale University, 51 Prospect Street, New Haven, CT 06525, November 1998.

[2] W. Daniel Hillis. U.S. Patent 5,212,773, 1993.

[3] Bradley C. Kuszmaul. Fast, deterministic routing, on hypercubes, using small buffers. *IEEE Transactions on Computers*, 39(11):1390–1393, November 1990.

[4] F. Thompson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.