Chapter 5
Data Path of Individual Components

The data path of an architecture defines the alignment of processing elements. It realizes a certain functionality which fulfills a given throughput constraints. There exit always different possibilities to realize a defined functionality. Here we describe the individual steps to the design of individual components, again tailored for communications systems as shown in Fig. 4.1. Note, there is a difference of a data path for individual components and the data path for processors. The data path of a general purpose processor has the task to be as flexible as possible, thus arithmetics units are used with large bit width to provide a high flexibility, e.g. 32 bit or 64 bit. For dedicated hardware design the data path and thus the computational units are optimized with respect to the given algorithm to be processed. The steps to derive a dedicated data path are:

1. Starting from an algorithm description we have to decompose the algorithm in functional parts. Goal of this task is to identify the mandatory processing. The processing can often be separated in individual processing steps. For the individual processing parts we have to analyze:
   - the final bit width of the data, e.g. what is the influences of the data representation on the communication performance,
   - how to realize processing kernels, e.g. processing in time or frequency domain,
   - the correct or approximative functionality, e.g. whether the communications performance of the approximation is good enough.
2. The next step is to derive a data flow. Deriving a data flow utilizing the complete functionality of the algorithm can sometimes be cumbersome. We have to distinguish between important aspects for the data flow and unimportant ones. The data flow is independent from the processing itself, rather reflects:
   - data dependencies, e.g. which data have to processed first,
   - lifetime analysis of data, e.g. how long do we have to hold information,
   - concurrency, e.g. the choice of consuming one value after the other or processing all at once.

F. Kienle, *Architectures for Baseband Signal Processing*,
DOI: 10.1007/978-1-4614-8030-3_5, © Springer Science+Business Media New York 2014

The data flow analysis is very important for the hardware realization since it directly influences the resulting throughput and the overall data handling. The data flow analysis should be done on an simplified or abstracted functional model. An example is shown in next section.

3. Define the constraints to derive a data path. For dedicated hardware design the major constraint is often a time budget to fulfill a certain task. One constraints could be the number of data which should be processed within a certain time budget which is in fact the throughput definition of an application.
4. Define a feasible data path of one possible realization which fulfills the timing budget/throughput constraint. The timing information can be an abstract timing or a detailed cycle based timing. Important is that we allocate the time budget for the processing task.

In the following we derive first an abstract data flow followed by different data path possibilities. We use the example of a Max-Log-MAP processing which was introduced in Sect. 3.3.3. The corresponding functional processing parts are derived afterwards. The data paths derived in the next section are valid for all forward-backward based algorithms, thus independent of a possible Log-MAP or Max-Log-MAP implementation. This is only possible if a clear separation of data flow and functionality is derived. Realizing corresponding processing elements are presented Sect. 5.2. More complicated functions can either be done by look-up tables or by deriving approximations which can be efficiently implemented. Both techniques are explained based on exemplary functions which are often used to realize channel decoders.

5.1 Data Flow to Data Path Example

The MAP decoder has to calculate the maximum a posteriori probability which was introduced in Sect. 3.3.3. We deal here only with the so called Max-Log-MAP algorithm which is typically employed in the hardware realizations. To perform the Max-Log-MAP algorithm on the trellis we have to perform the so called forward-backward processing, this is shown step-by-step at Figs. 3.10–3.14.

To separate the processing and the data flow we first derive the mandatory processing steps in a more general way. The Max-Log-MAP decoding according to Sect. 3.4.2 can be summarized in four processing steps which are repeated here.

- Branch metric computation and allocation on the edges of the trellis.
- Forward recursion: calculate at each time step and thus trellis step k the corresponding path metric α_{k+1}^{S} for every state S. The calculation is done recursively, where the result in step k always depends on the previous results.
- Backward recursion: the backward recursion is based on exactly the same recursive processing, however starting with the calculation from the last state in the trellis.

- Soft-output calculation: to calculate the symbol-by-symbol MAP probability we need the results from the forward recursion, the backward recursion and the branch metrics.

The recursive calculations are always done on multiple values. In the case of trellis processing we calculate at each recursion step $\boldsymbol{S}_{k+1} = f(\boldsymbol{S}_k, \boldsymbol{\gamma}_k)$. In the case of a 4-state trellis $\boldsymbol{S}_{k+1}$ consist of 4 values, while $\boldsymbol{\gamma}_k$, the edge labels, consists as well of 4 branch metrics.

For the data flow we do not concern about the cardinality of $\boldsymbol{S}$ or $\boldsymbol{\gamma}_k$ and we can abstract the recursion function by a simple function which calculates only one value,

$$S_{k+1} = f(S_k, \gamma_k). \tag{5.1}$$

Each branch metric at step k is assumed as well to be one value. For deriving the data flow we use a block of 8 input values, denoted as $\boldsymbol{\gamma} = [a, b, c, d, e, f, g, h]$, with e.g. $\gamma_0 = a$ at step $k = 0$. Furthermore, for the function of Eq. 5.1 we utilize a simple minimum search of two input values, i.e. $\min(x, y)$. The entire forward-backward processing is thus reduced to a simple processing of 8 input values with simple computational stages.

The abstract function for deriving the data flow can be written as a marginalization problem:

$$M(\boldsymbol{x}) = \min_{\sim x_i}\{a, b, c, d, e, f, g, h\} \tag{5.2}$$

$\sim x_i$ defines the values of $\boldsymbol{x}$ without the value at position i. Thus, we have to compute 8 different results, one for each input position.

$$\begin{aligned}
M(x_i = a) &= \min\{b, c, d, e, f, g, h\} & M(x_i = b) &= \min\{a, c, d, e, f, g, h\} \\
M(x_i = c) &= \min\{a, b, d, e, f, g, h\} & M(x_i = d) &= \min\{a, b, c, e, f, g, h\} \\
M(x_i = e) &= \min\{a, b, c, d, f, g, h\} & M(x_i = f) &= \min\{a, b, c, d, e, g, h\} \\
M(x_i = g) &= \min\{a, b, c, d, e, f, h\} & M(x_i = h) &= \min\{a, b, c, d, e, f, g\}
\end{aligned}$$

One possible data flow structure to solve Eq. 5.2 is a tree structure. The tree structure is a typical data flow with respect to a parallel realization.

Parallel means here that we may process all input values $\{a, b, c, d, e, f, g, h\}$ in one step.[1] The tree structure for two functions can be seen in Fig. 5.1. The left tree calculates $M(x_i = a)$ and the right tree $M(x_i = b)$ respectively. It can be seen that in both trees identical sub-results are calculated, indicated by the circles. This gives us an indication about possible hardware reuse in a derived data path.

However, first we define additional constraints on our data flow with respect to data dependencies. The MAP processing is recursive and non commutative, at least when utilizing the trellis structure of convolutional codes. We have to ensure a recursive approach as well to solve Eq. 5.2. This is shown in the new data flow structure of

[1] In a hardware realization this would mean in the same clock cycle.

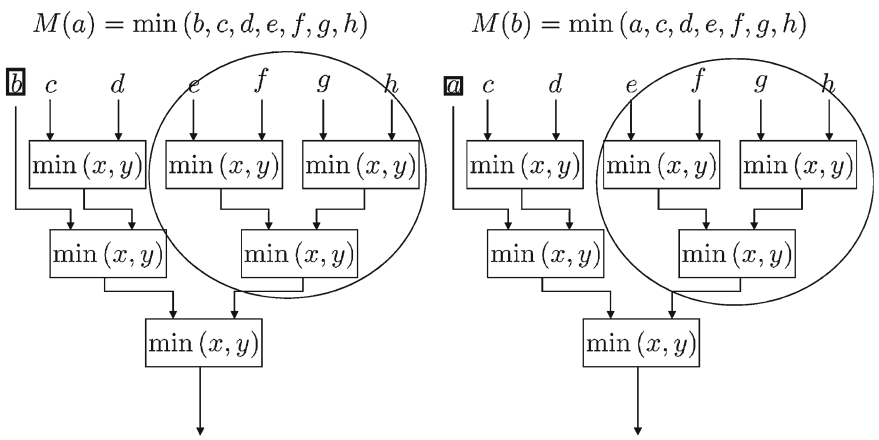


Fig. 5.1 Tree data flow for calculating $M(x_i = a)$ and $M(x_i = b)$

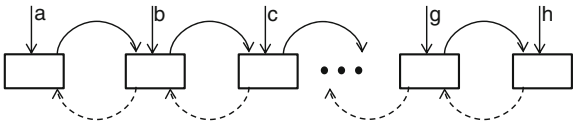


Fig. 5.2 Recursive data flow to calculate $M(x_i = a)$ and $M(x_i = b)$

Fig. 5.2. Two different data flow directions are shown here, the forward direction with solid lines, and the backward direction with dotted lines.

In the following we derive two different data paths which fulfill the constraint of a recursive calculation. The data path contains already timing information and is one major step towards a dedicated hardware architecture. Then, one processing step in the calculation corresponds to one clock cycle.

5.1.1 Serial Data Path: One min *(x, y) Unit*

The serial data path (serial marginalization) has already prerequisites for a final hardware realization. Here in the example, we assume that we can process one input value per clock cycle (processing step). Figures 5.3–5.5 show the data path and the processing steps to solve Eq. 5.2. We write all input information (a, b, d, e, f, h, g) to an array, indicated as box, which may be later instantiated as a memory. The current reading position of the array is indicated as a pointer. In the hardware implementation this pointer will be a reading address for a RAM, see Sect. 4.2. Each processing step is now described step by step.

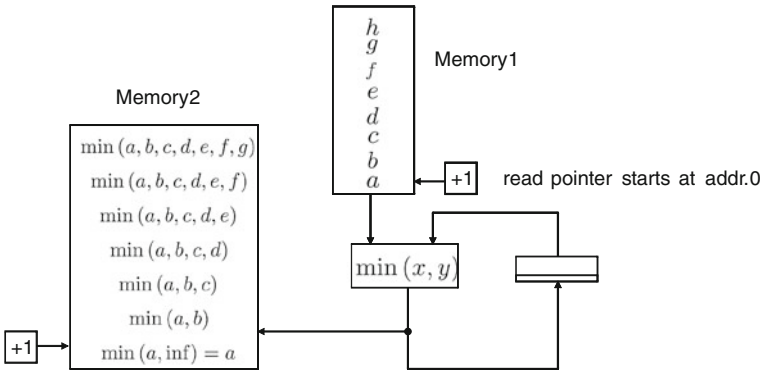


Fig. 5.3 Serial architecture: forward recursion, with the content of memory 2 after the 7th clock cycle

Step 1: Forward Processing (Figure 5.3)

- Initialization:
  The input values $\{a, b, c, d, e, f, g, h\}$ are stored sequentially in memory 1.
  The read pointer (address) for memory 1 is set to 0 (the address where value a is stored).
  The write pointer(address) for memory 2 is set to 0.
  The register is initialized with an 'infinite value', e.g. in hardware this would be the largest number with respect the utilized number representation.
- First clock cycle:
  Read the first value (a) from memory 1.
  $\min(x, y)$ calculates the first result which is $\min(a, \inf) = a$.
  The result is stored in memory 2.
  The register keeps now the value a.
  The read and write pointer are incremented by one.
- Second clock cycle:
  Read the second value (b) from memory 1.
  $\min(x, y)$ calculates the second result which is $\min(a, b)$.
  The result is stored in memory 2.
  The register keeps now the value $\min(a, b)$.
  The read and write pointer are incremented by one.
- 7th clock cycle
  After 7 clock cycles all intermediate results are stored in memory 2.
  The so called **forward processing** or **forward recursion** is finished.

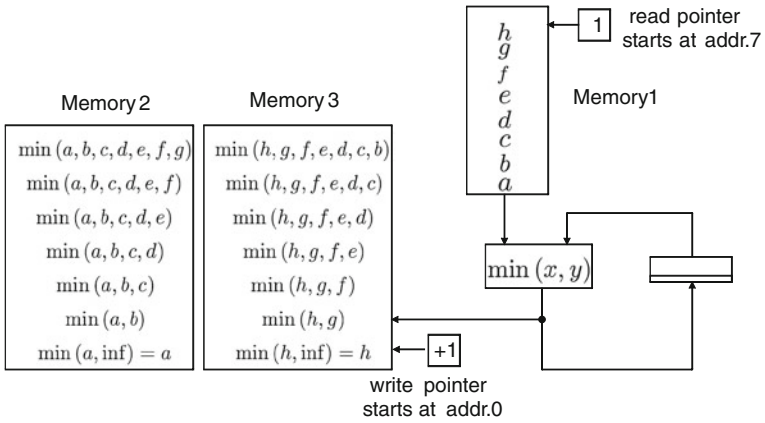


Fig. 5.4 Serial architecture: backward recursion, with the content of memory 3 after the 7th clock cycle

Step 2: Backward Processing (Figure 5.4)

For the backward processing a third memory has to be instantiated, denoted as memory 3.

- Initialization:
  The input values $\{a, b, c, d, e, f, g, h\}$ are again stored sequentially in memory 1.
  The read pointer (address) for memory 1 is set to 7 (the address where value h is stored).
  The write pointer(address) for memory 3 is set to 0.
  The register is initialized with an infinite value.
- First clock cycle to last clock cycle:
  Read the memory 1 starting form address 7, first value is h, last is a.
  The $\min(x, y)$ calculates the results in reversed order.
  The register keeps always the intermediate values.
  The read pointer of memory 1 is decremented by one in each step. The write pointer of memory 3 is incremented by one in each step.

Step 3: Output Processing (Figure 5.5)

For the output processing the data passed to the $min(x, y)$ are multiplexed from memory 2 and memory 3 respectively. The two new introduced multiplexers indicate the required switching.

- Initialization:

  The read pointer (address) for memory 2 is set to 0.
  The read pointer(address) for memory 3 is set to 6.

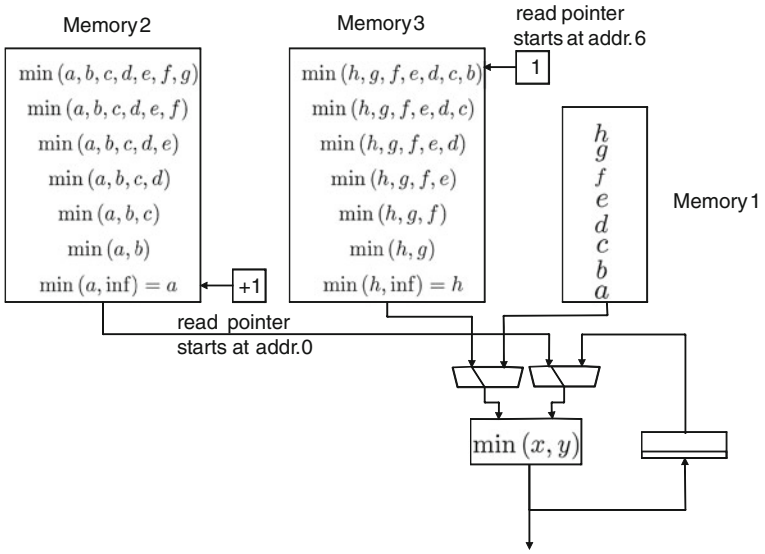


Fig. 5.5 Serial architecture: backward recursion

- First clock cycle:
  Only the value from memory 3 at address 6 is read which is already the first result $M(a) = \min(h, g, f, e, d, c, b)$.
  The pointer of memory 3 is decremented by one.

- Second clock cycle:
  Read address 0 from memory 2.
  Read address 5 from memory 3.
  $min(x, y)$ calculates the next result $M(b)$. The pointer of memory 2 is incremented by one.
  The pointer of memory 3 is decremented by one.

- Last clock cycle:
  Only the value from memory 2 at address 6 is read, this is already $M(h) = \min(a, b, c, d, e, f, g)$.

Summary data path: one min (*x*,*y*) unit

For the processing we instantiate three memories and one processing unit. One memory stores the input values, memory2 and memory 3 store the intermediate results of the forward, and backward processing respectively. The overall number of clock cycles for the processing can be decomposed into three processing steps which are: 7 clock cycles for the forward processing, 7 clock cycles for the backward processing and 8 clock cycles for the output processing. Thus, the overall number

of cycles is 22 to calculate 8 output values. On average, one output value needs $\#cycles/value = 22/8 = 2.75$ cycles. The presented data path utilizes a so called resource sharing of one processing unit, while load of the processing unit is 100 %.

5.1.2 *Serial Data Path: Two* min(*x,y*) *Units*

The section shows a serial data path using one additional $\min(x, y)$ unit. The goal here is to trade off processing units versus memory usage. The basic difference to the first approach is the simultaneous processing of the backward and output processing. The processing is now performed in two processing steps which are again described step-by-step:

Step 1: Forward Processing Figure 5.3

The forward processing remains identical to the description of Fig. 5.3.

Step 2: Backward Processing and Output Processing (Figure 5.6)

- Initialization:
  The input values $\{a, b, c, d, e, f, g, h\}$ are stored sequentially in memory 1.
  The read pointer (address) for memory 1 is set to 7 (the address where value h is stored).
  The write pointer(address) for memory 2 is set to 6 (the address where the last forward result is stored).
  The register is initialized with an infinite value.
- First clock cycle:
  Read the value (h) from memory 1.
  The first min-search unit calculates the first result which is $\min(h, \inf) = h$.
  The result is stored in the register, the register keeps the value h.
  Read the last forward processing result from memory 2, which is min (a, b, c, d, e, f, g).
  The second min-search unit calculates the first output result $M(h)$.
  The read pointer of memory 1 and 2 is decremented by one position.
- Second clock cycle:
  Read the next value (g) from memory 1.
  The first min-search unit calculates the second result which is $\min(h, g)$.
  The result is stored in the register.
  The second min-search unit calculates the second output result $M(g)$.
  The read and write pointer is decremented by one.
- Last clock cycle:
  The last result $M(a)$ is stored in the register after the 7th clock cycle.

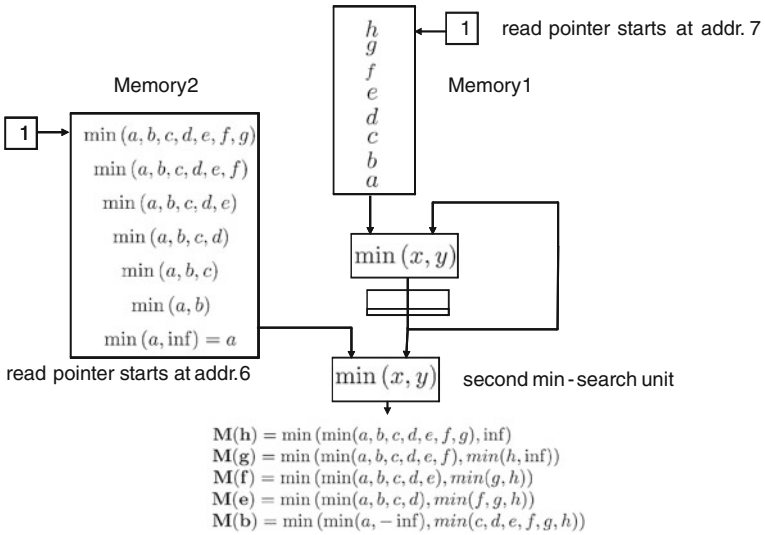


Fig. 5.6 Serial architecture: simultaneous backward and output processing

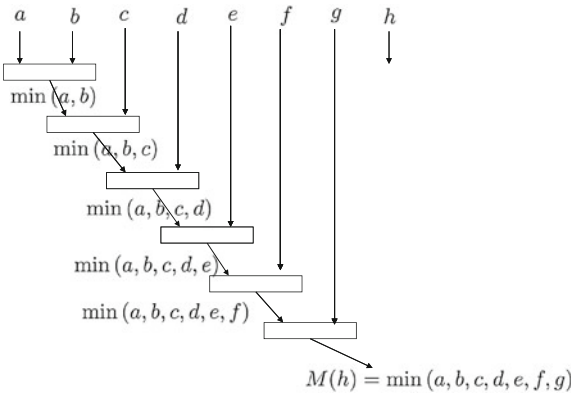
No additional reading from memory 2 is mandatory.
The backward processing including the output processing is finished.

Summary data path: two min (*x*,*y*) units

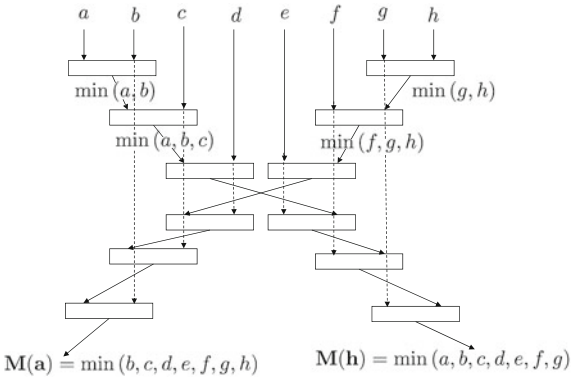
For the processing we instantiate two memories and two processing unit. One memory to keep the input values, and one memory to store the intermediate result of the forward processing. The overall number of clock cycles for the processing can be decomposed into two processing steps which are: 7 clock cycles for the forward processing, and 8 clock cycles for the backward recursion and output processing. Thus, the overall number of cycles is 15 to process 8 output values. On average, one output value needs $\#cycles/value = 15/8 = 1.875$ cycles. The presented serial data path puts focus on a simultaneous processing of different tasks. Two important difference can be seen when comparing the architecture with one and two $\min(x, y)$ unit. First, the result is obtained in a different order, second, the number of required clock cycles is much lower for the architecture according to Fig. 5.6.

5.1.3 Data Path: Parallel Processing

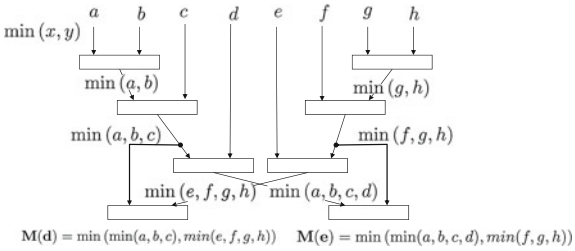
Parallel processing means, that more than one input value is processed in every clock cycle. Since we still have to solve Eq. 5.2 in a recursive manner this is not trivial. The architecture for a possible parallel processing is derived step-by-step.



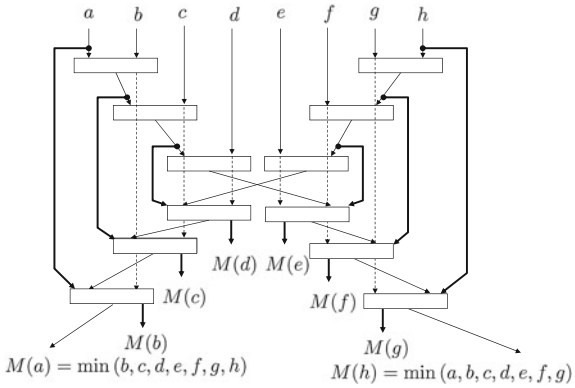
The first step is the so called unrolling of the calculation of $M(h)$. Here, we instantiate six functional units ($\min(x, y)$), which are shown as boxes in this figure Each unit receives the result of the previous unit and one of the input values as input. The intermediate result are written next the outputs of each unit.



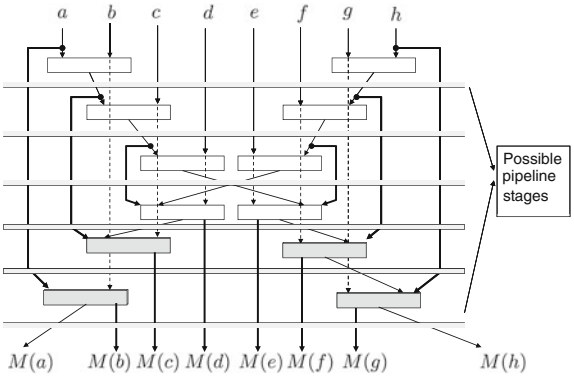
This figure shows the unrolling of the recursions from the beginning and the end. Two results are obtained $M(a)$ and $M(h)$. Every input value is consumed by two $\min(x, y)$ functions. The dotted arrows indicate that a certain value is also an input to a second functional units. For better readability some of the intermediate values are not labeled in this figure.



Here, we use two intermediate results to calculate other $M(x_i)$ values. For example $M(d)$ is obtained by an additional $\min(x, y)$ unit, the inputs x and y are the intermediate result $x = \min(a, b, c)$ which is obtained from the forward recursion, and $y = \min(e, f, g, h)$ which is calculated by the backward recursion respectively.



This figure shows the full parallel processing of all results. Every intermediate result, as well as every input result is used two times. Here, the data dependency restricts the order in which the operations can be performed. Note also that every functional unit in gray features tow instances of $\min(x, y)$ units. One for the output processing and one to pass an intermediate result to the next recursion stage. It is worth noting the similarities to the serial output processing of Fig. 5.6.



Typically we can not process all functions in one clock cycle. Hence, additional registers have to be instantiated which are called pipeline stages. At each pipeline stage we have to store all values which are consumed at a later time step. When ever a line of the derived data flow is crossing the horizontal pipe line stages the corresponding values have to be stored. It can be seen that the number of values we have to store varies for each pipe line stages.

Summary data path: parallel processing

In the following we assume that pipeline stages are instantiated after each processing block.

- The parallel architecture has a latency of 6 clock cycles, before the first output is valid, then in each clock cycle an **entire** block is calculated.
- We need 6 forward and 6 backward recursion units ($\min(x, y)$ units) and 6 output units.
- $(N-1) \cdot (N-1)/2$ intermediate results have to be stored, with N=8.
- In every clock cycle we have to read all 8 input values
- After the pipeline is filled we need one clock cycle to calculate all 8 results, the required cycles to process one value is thus $\#cycles/value = 1/8 = 0.125$ cycles.

The question whether to use a serial or parallel architecture leads to the question of required throughput, data storage problems and data access problems. The requirements for providing the correct input data at the right time is completely different for both architectures. Furthermore one has to analyze the complexity when utilizing the full functional unit of our application. Thus we have now to replace the $(\min(x, y))$ by the correct processing units, which leads to the so called recursion units.

5.2 Deriving Processing Units

In the previous section we have derived the data path for serial and parallel processing. As an example the data flow of a general forward-backward algorithm was used. As mentioned the data path defines the alignment of processing units. These procession

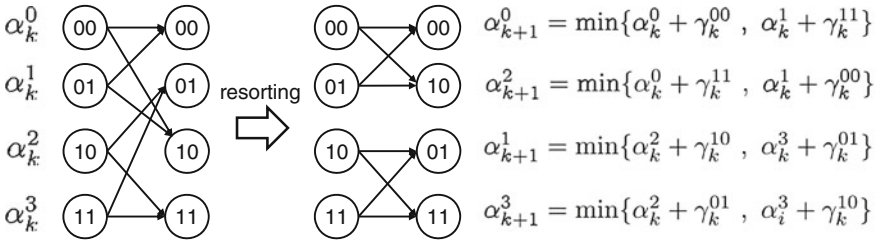


Fig. 5.7 Calculations of one trellis step.

units have to be designed with respect to the required functionality of the application. The functional units for the Max-Log-MAP algorithm are derived in Sect. 5.2.1. Often we have to perform algorithmic manipulations to realize the correct functionality. Either we can use look-up table to realize the mandatory functionality or we can derive good approximations by algorithmic transformation. Both techniques are shown by examples in Sects. 5.2.2 and 5.2.3, respectively.

5.2.1 Recursion Units

In this section we derive the processing units for the Max-Log-MAP algorithm. It is quite obvious that the forward recursion and the backward recursion share identical functionality, which can be seen by comparing the two mandatory equations for α and β processing (Eq. 5.3).

$$\alpha_{k+1}^{m'} = \min_{\forall(m->m')} \left(\alpha_k^m + \gamma_k^{x_i,x_i+1}(S_k^m, S_{k+1}^{m'})\right)$$
$$\beta_k^{m'} = \min_{\forall(m'->m)} \left(\beta_{k+1}^{m'} + \gamma_k^{x_i,x_i+1}(S_k^m, S_{k+1}^{m'})\right) \tag{5.3}$$

These equations where derived in Sect. 3.4.2. An unit implementing the functionality of this recursion step is called recursion unit (RU). It is always a function of the old state metrics and the input branch metrics. Figure 5.7 shows one example trellis with 4 states, while the trellis on the left reflects one trellis step. Starting form this example we derive a corresponding recursion unit. In trellis step k each state has two possible state transitions to a successor state in trellis step $k+1$. The trellis transition can be mapped to so called butterflies, however, the output has to be resorted. A butterfly represents the processing of two input to two output states, the direction (forward or backward) itself is not of importance. The advantage of the decomposition in butterflies and resorting is the clear separation of a regular processing part and a resorting of the state results.

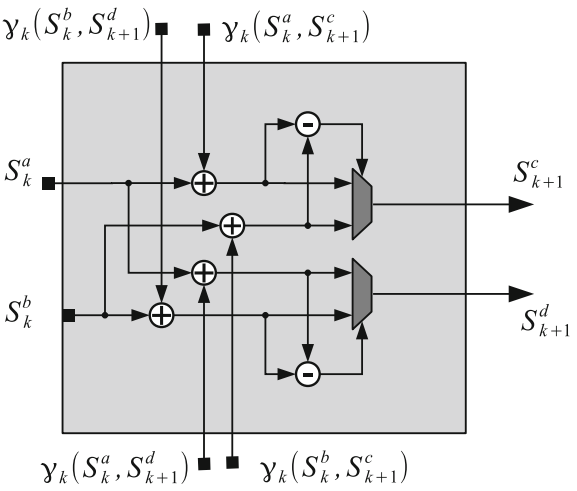


Fig. 5.8 Butterfly architecture for Viterbi decoding or Max-Log-MAP decoding

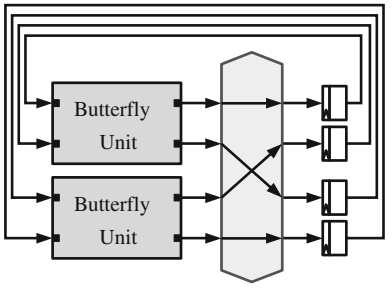


Fig. 5.9 Full recursion unit to process a 4-state trellis. A realization to process a larger amount of states can be done by an appropriate number of instantiated butterfly units.

A butterfly unit for Max-Log-MAP processing or Viterbi decoding is shown in Fig. 5.8. Every Butterfly unit is composed of two $\min(x, y)$ units simply realized as compare select units (CS). The CS part is realized by one subtraction, while the sign of the result is used as the multiplexer control signal.

The inputs for the butterfly units are the old states $S_k^{a,b}$ and the corresponding branch metrics which connect the corresponding states, e.g. $\gamma_k(S_k^a, S_{k+1}^c)$ represents the path metric between state S_k^a and S_{k+1}^c. Since one butterfly unit processes four state transitions we have four branch metrics as input. Note that for a convolutional code of $R = 1/2$ typically only two distinct branch metrics exist, then $\gamma_k(S_k^a, S_{k+1}^c) = \gamma_k(S_k^b, S_{k+1}^d)$ and $\gamma_k(S_k^b, S_{k+1}^c) = \gamma_k(S_k^a, S_{k+1}^d)$ respectively.

The recursion unit itself is implemented as multiple instances of the butterfly units and shuffling unit. The RU unit presented here can process one trellis step in one clock cycle. This is shown for a 4-state code in Fig. 5.9. The old state metrics are read from the registers and the new updated states are written to the registers. This can be done within one clock cycle. The shuffling unit has to ensure that the original trellis connectivity structure is preserved. The derived structure with four states can be extended for more states which is done by increasing the number of butterfly

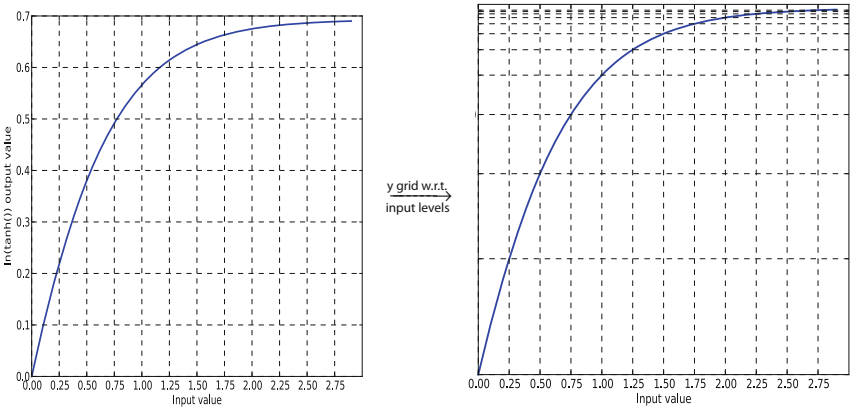


Fig. 5.10 Lookup table analysis and generation. Left shows the plot with a linear grid of the y-axis, right shows the y-axis with a grid with respect to the quantized input values

instances. The merge of the derived functional unit with one derived data path will result in a detailed Max-Log-MAP architecture. One specific instance of a so called serial MAP architecture will be shown in Sect. 6.3.1 in the context of building turbo code decoders.

5.2.2 Look-Up Tables

Sometimes it is difficult to realize a function in hardware. If the input bit width or the output bitwidth is small we can realize the corresponding function by a look-up table. A look-up table stores pre-computed values for each possible input combination. Deriving such a look-up table is shown Fig. 5.10. As an example we derive the function

$$f(x) = ln(tanh(x)). \tag{5.4}$$

This equation is used for processing so called check nodes, as introduced in Chap. 7, here it is used for demonstration purposes only. The left plot shows the function of $ln(tanh(x))$ with the input value on the x-axis, the function value on the y-axis. For the look-up table we first have to evaluate the input quantization. Here, the right plot features an input granularity of 0.25. That means the input values have two bits for the fractional part. Now we have to derive the corresponding output value. Due to the non-linear function the output value requires a higher resolution. For large input values we can barely distinguish the output results. For the look-up table we have to decide on how many bits to represent the corresponding y values. Communications performance simulations are mandatory to see if the chosen granularity will result in a performance degradation.

Look-up tables are simple to realize in hardware, however, the size of the values to be stored may quickly come infeasible for a large input width. Then, more elaborated techniques like an additional compression or the direct approximation of the function may become mandatory.

5.2.3 Deriving an Approximation Function

Approximative functions are used when a direct implementation will not fulfill the throughput constraints, results in a large area, or shows a high power consumption, respectively. For example realizing functions with many input variables quickly increases the number of stored values when realizing these by look-up tables. Thus, we should use algorithmic transformations to see whether it is possible to find a more suitable form for implementation. Often we can only approximate the desired function due to given constraints, however, for many applications this will be good enough. There exist no general rule how to derive a suitable approximation for an efficient hardware realization. Rather, the designer often relies on many different mathematical techniques and its own experience.

In the following one example is presented to derive an approximation suitable for hardware realization. The symbol-by-symbol MAP processing of a single parity check code is used for demonstration. The equation for the mandatory processing was derived in Sect. 3.3.3. Assuming a single parity check code with three bits we have to process:

$$\lambda_q = \ln\left(\frac{e^{\lambda_p} e^{\lambda_r} + 1}{e^{\lambda_p} + e^{\lambda_r}}\right) \tag{5.5}$$

This equation can be derived from Eq. 3.30.[2] It shows the output calculation for bit message q utilizing the two input messages p and r. In the first transformation step we utilize the already introduced Jacobian logarithm in its positive form

$$\ln\left(e^{\delta_1} + e^{\delta_2}\right) = \max^*(\delta_1, \delta_2) = \max(\delta_1, \delta_2) + ln(1 + e^{-|\delta_1 - \delta_2|}).$$

Using the Jacobian logarithm in Eq. 5.5 results in:

$$\lambda_q = \ln\left(\frac{e^{\lambda_p} e^{\lambda_r} + 1}{e^{\lambda_p} + e^{\lambda_r}}\right) = \max^*(\lambda_p + \lambda_r, 0) - \max^*(\lambda_p, \lambda_r) \tag{5.6}$$

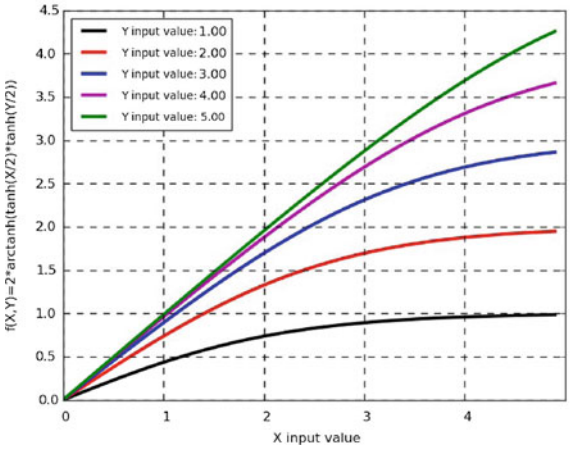
The result can be further decomposed in two maximum searches and two parts determining the correction term.

[2] The function can equivalent expressed as $\lambda_q = 2 \cdot \operatorname{arctanh}\left(\tanh\left(\frac{\lambda_p}{2}\right) \cdot \tanh\left(\frac{\lambda_r}{2}\right)\right)$. This trigonometric expression is sometimes used in literature.

$$\lambda_q = \underbrace{\max(\lambda_p + \lambda_r, 0) - \max(\lambda_p, \lambda_r)}_{approximation\ \tilde{\lambda}_q} + \underbrace{ln(1 + e^{-|\lambda_p + \lambda_r|}) - ln(1 + e^{-|\lambda_p - \lambda_r|})}_{correction\ term\ \delta}$$

This is an important result since we have clearly separated a simple approximation part and an additional correction part. Table 5.1 shows results for λ_q assuming different input values. The true result is decomposed in an approximative output $\tilde{\lambda}_q$ and a correction term δ . The approximation term is always the minimum of the absolute values, while the sign ensures that the single parity check condition is fulfilled. The largest correction term results when both input values are identical.

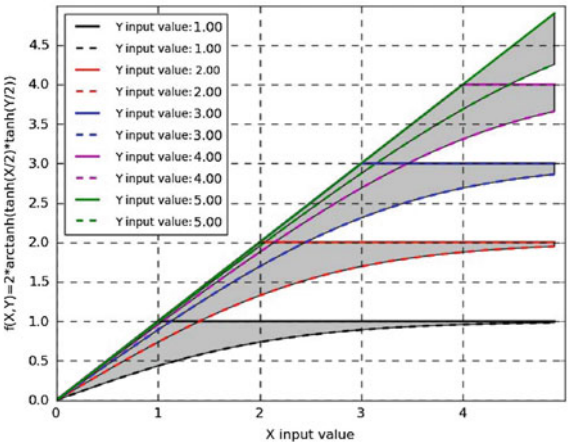
In the following the visual evaluation of Eq. 5.5 is shown, together with a possible realization of the correction term. The visualization often helps to get an idea about the quality of approximations. The used approximation derived within the following four figures was originally presented in [1].



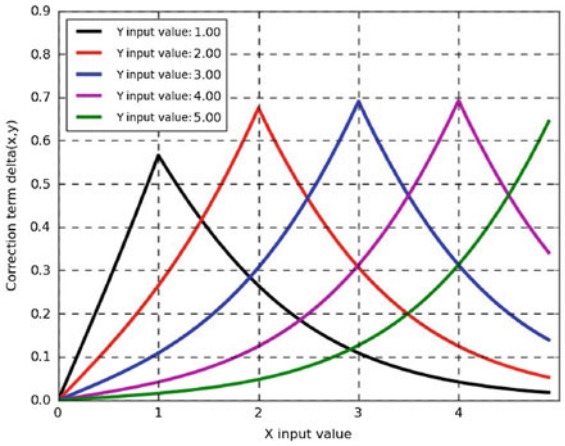
Plotted is Eq. 5.5, here denoted as $f(x, y) = 2 \cdot \text{arctanh}\left(\tanh\left(\frac{x}{2}\right) \cdot \tanh\left(\frac{y}{2}\right)\right)$. The x-scale represents the input value x, while 5 different y values result in 5 different graphs. The larger the y value, the larger the output result (y-axis). Thus, y = 1 will be always the graph at the bottom, y = 5 at the top of the graph. This is true for all following figures.

Table 5.1 λ_q calculations and approximation $\tilde{\lambda}_q$ for different input values

Input 1 λ_p	Input 2 λ_r	Approx. $\tilde{\lambda}_q$	Correction δ	Output λ_q
+5	+2	+2	−0.05	+1.95
+2	+5	+2	−0.05	+1.95
+5	−2	−2	+0.05	−1.95
−2	+5	−2	+0.05	−1.95
−5	−2	+2	−0.05	+1.95
−2	−2	+2	−0.68	+1.32
−5	−5	+5	−0.68	+4.32

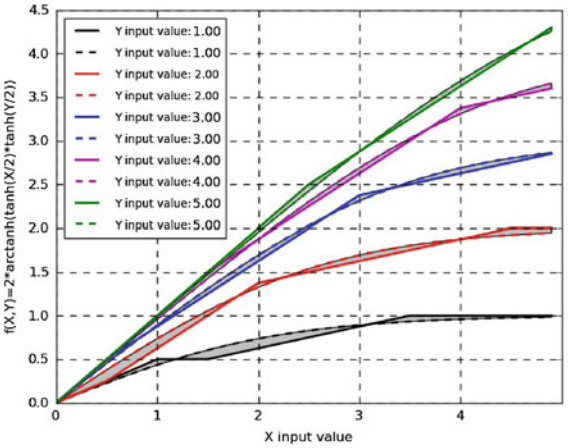


Plotted is the approximation $f(x, y) \sim \min\{|x|, |y|\}$ (dotted lines) and the original function. The shaded area highlights the difference between the true result and the approximation. It can be seen that the approximation error is always the largest for $x = y$.



The difference between the original function and the min approximation is shown. Thus, the correction function is plotted which is $\delta(x, y) = ln(1 + e^{-|x+y|}) - ln(1 + e^{-|x-y|})$ The left graph has $y = 1$ as its input value, while the right most plot has a constant

$y = 5$ value. A hardware approximation of this function can be derived by a Taylor approximation which is the result presented in the next step.



Here the original function and the final utilized approximation is shown. The hardware approximation term utilized here is: $\delta(x, y) \sim \psi(|x + y|) - \psi(|x - y|)$ With a hardware friendly ψ function.

$$
\begin{aligned}
&function\,\psi(a):\\
&\quad d = \frac{5}{8} - \frac{a}{4}\\
&\quad if\,(d > 0.0):\\
&\qquad return\,d\\
&\quad else:\\
&\qquad return\,0.0
\end{aligned}
$$

Comparing the final result one can see that the difference is rather small. This approximation can be applied in a recursive manner for the serial computation of single parity check codes with more bits. For example assuming three bits instead of two, the approximation can be calculated according to $\delta(x, \delta(y, z))$.

The solution derived in this example is only one possible approximation of Eq. 5.5. Providing the accurate functionality with respect to a given frequency constraint is the major challenge when deriving a possible realization. If the cycle budget can not be fulfilled, either we have to find a different approximation, or we have to increase the parallelism of the data path to allow for a reduced frequency constraint.

Final architectures featuring data path and processing units for turbo decoders and LDPC decoders are shown in Chaps. 6 and 7. For both type of decoders the respective design space is described highlighting different algorithmic and architectural possibilities.

Reference

1. Mansour, M.M., Shanbhag, N.R.: High-throughput LDPC decoders. IEEE Trans. Very Large Scale Integr. Syst. **11**(6), 976–996 (2003)