# GDA Implementation Requirements

**Document No:** **GP-ASG-TN-002**

**Issue No.:** **1**

**Rev. No.:** **-**

**Issue Date:** **18.06.03**

| | Name | Date |
|---|---|---|
| prepared by | Astrium GmbH PDHE Team | 18.06.2003 |
| approved by | C. Schaefer, SE71 | 18.06.2003 |
| authorized by | | |

## CONTENTS

# 1 DOCUMENTS

## 1.1 Applicable Documents

AD-1 GAIA/TADG21/2001 - GAIA Payload Data Handling Electronics, Statement of Work, Issue 1, Rev. B, 09/10/01, incl. Annex 1, Technical Requirements, modified by MoM of PDHS Negotiation Meeting 17/12/02

AD-2 OBD-FC-01, Gaia_Detect description, rev. 1.4, Dec 11, 2002

AD-3 OBD-CoCo-03, Input to PDHE contract, Version 1.5, March 21, 2003

AD-4 OBD_SM_02, Flowcharts of the SWA 2.5 and GD 1.5 detection algorithms, March 14, 2003

AD-5 OBD-HC-01, First implementation of the selection algorithm, V 1.1, November 25, 2002

AD-6 GAIA-CUO-117, Scientific requirements for the on-board processing, March 31, 2003

AD-7 OBD-CoCo-06, Comments on GP-ASG-RS-0001 implementation analysis, May 7, 2003

AD-8 OBD_SM_03, An improved deblending scheme for GAIA's on-board detection, March 5, 2003

## 1.2 Reference Documents

RD-1 EF5/FR/PC/038.02 - GAIA System Level Technical Reassessment Study, Final Report, Issue 2, 27/06/02

RD-2 GP-ASG-RS-0001, Analysis of GAIA star detection and selection algorithms from implementation point of view, March 26, 2003

# 2 INTRODUCTION

The purpose of present document is to assess the requirements for implementing the Gaia Detect Algorithm (GDA) as specified in AD-2, AD-3, AD-4, and AD-5 for real-time performance under worst case sky density conditions. This assessment shall provide an estimate of the number of processing modules (FPGA, ASIC, microprocessors) required for a one-to-one implementation of the GD algorithm. The estimate will be input to the concurrent ASTRO-VPU architecture design, eventually leading to the expected VPU budgets.

A first estimate of a pure software implementation in RD-2 has led to a prohibitive micro-processor load. A hardwiring of certain functions is therefore most desirable, and has indeed been required in AD-1. The GDA functions can be decomposed into pixel-based, clock-driven, low-level functions on one hand, and object-based, data-driven, high-level functions on the other. At the GAIA PDHE Progress Meeting 1 (May 21, 2003) consensus has been established that the object-based functions, due to their complexity, are not proper candidates for hardwiring. On the other hand, despite their relative complexity, the pixel-based shall be considered for hardwiring. This approach has been taken presently and is the baseline for the present estimation of processing resources.

Due to the complexity of the object-based functions, the present theoretical software processing load estimate is only a ROM-estimate which needs to be confirmed. The most efficient way to get a reliable estimate is by implementation on a micro-processor with an instruction set similar to that of the Leon µP (which is the preferred candidate for an FM microprocessor).

# 3 OVERVIEW

The GD detection and selection algorithm has been functionally decomposed (Figure 3-1 and Figure 3-2). Blue boxes show functions to be implemented in hardware, yellow boxes in software. Solid lines represent VPU-internal data interfaces, dotted lines represent data interfaces with VPU-external physical units.

The individual subfunctions are analyzed in detail in the following sections. Presently, an overview is given, with particular attention being paid to data rates at module interfaces.

ASM Processing essentially incorporates GAIA detection for both channels. The detection algorithm can be decomposed into pixel-based and object based operations, called detection (1) and detection (2), respectively. The input to detection (1) is the complete pixel stream of either ASM. The data rate is

$$1360/2 \text{ Hz} \times 983 \text{ pixels} = 668440 \text{ pix/s} .$$

The output of detection (1) shall be only those pixels which have been detected by GD as elements of detected objects ("object member pixels" or OM-pixels). The according reduction factor in pixel rate has been estimated at >10, resulting in a data rate of <70000 pix/s. At this interface, however, for each OM-pixel the following data structure shall be output:

1. raw pixel value (16 bits)
2. filtered pixel value (16 bits)
3. background pixel value (16 bits)
4. SNR² for that pixel (32 bits)
5. object label for that pixel (16 bits)
6. pixel AC-position (1...983) (16 bits)

This results in a data rate of

$$< (5 \times 16+32) \times 70000 \text{ bits/s} = 7.5 \text{ Mbps}$$

Detection (2) operates at object level and is therefore proposed for implementation in software. Detection (2) receives OM-pixels line by line and accumulates objects. Detection (2) maintains a list of as many as 64 objects which are simultaneously being accumulated and processed. (The proposed data handling is not block-based and therefore avoids the fragmentation and merging of objects at block boundaries.) When an object has been completed - i.e. when the current line supplies no further connected pixels - the computation of object features is finalized and the object is stored in a data base (called LUT1). This data base is being filled at the maximum rate of

$$2 \times 1825 = 3650 \text{ obj/s}.$$

1825 objects/second is the maximum rate of objects detectable per telescope in crowded skies (AD-7). A factor of 2 has been added at this interface to accomodate for improper objects which may arise due to cosmic rays and due to a potential oversensitivity of the detection (1) algorithm. These improper objects will be eliminated by the AF1 selection algorithm which is part of the AF1 Processing block.

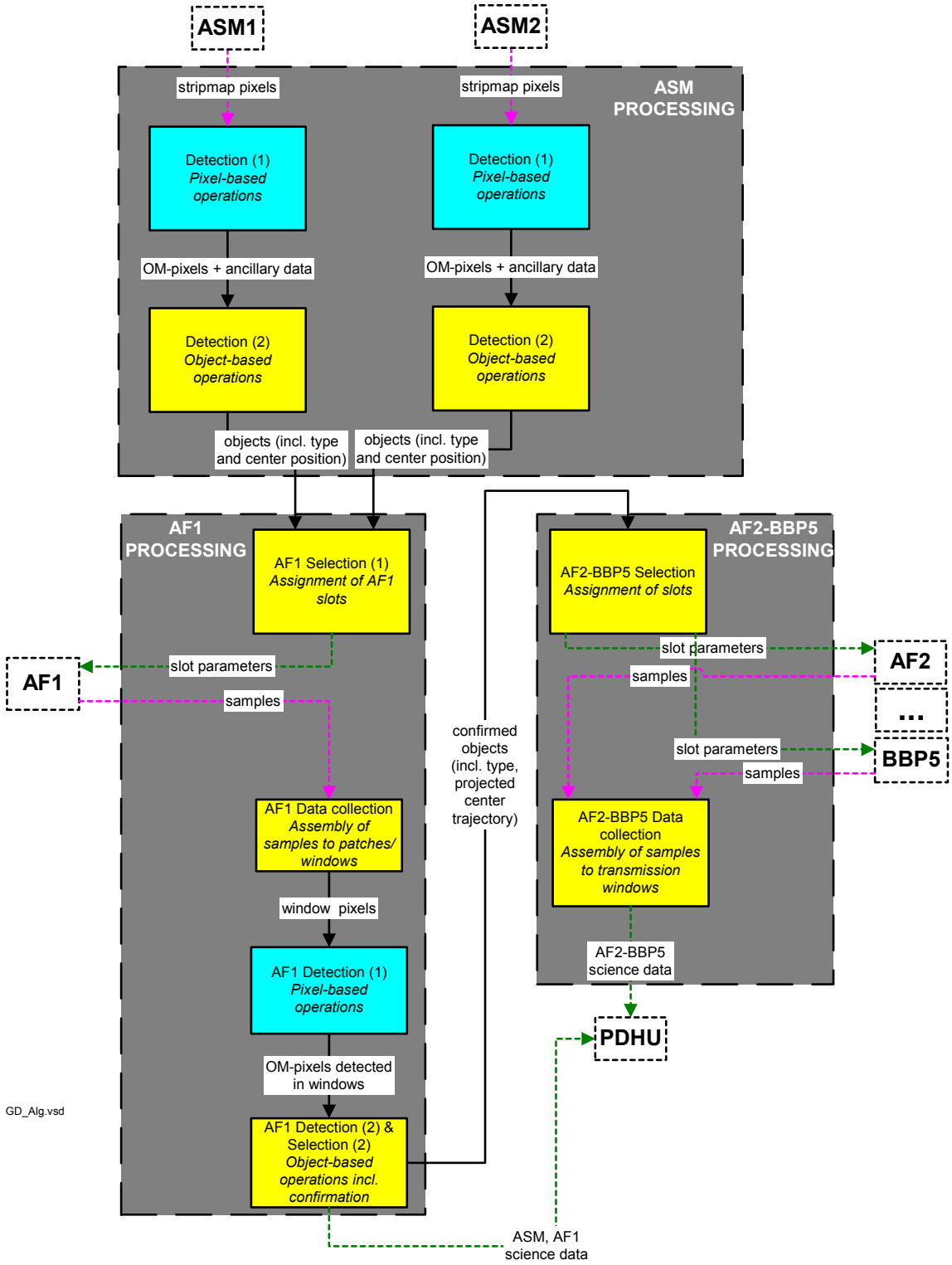


**Figure 3-1: Functional blocks for implementation of GD**

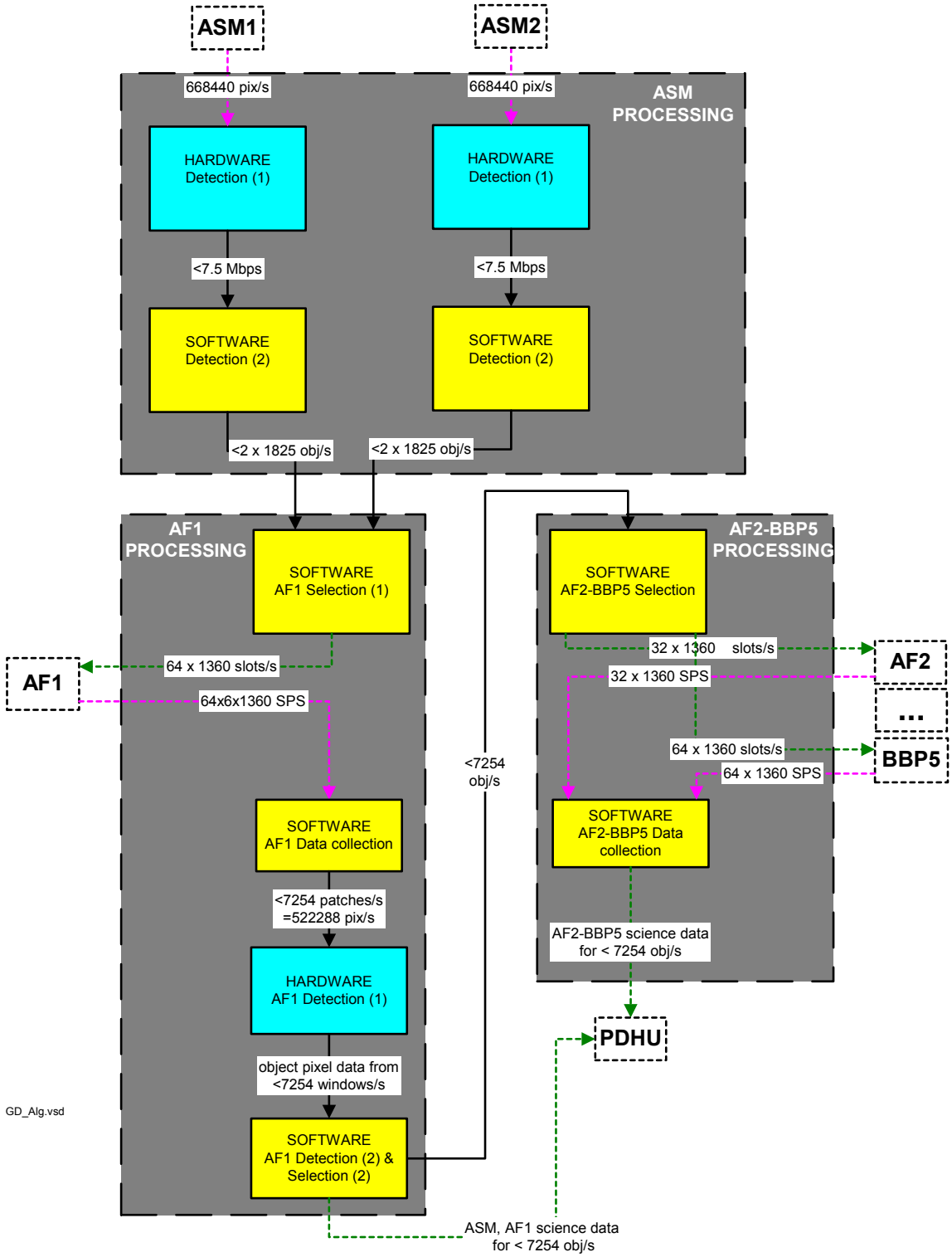


**Figure 3-2: Data rates at functional block interfaces**

AF1 Processing consists of four subblocks:

- Selection (1)
- Data Collection
- Detection (1)
- Detection (2) and Selection (2)

The objects retained after AF1 Selection (2) constitute the input to the final AF2-BBP5 Processing block.

The AF1 Selection (1) algorithm is fed with <7300 obj/s. AF1 slots are to be assigned by selection (1) for AF1 windowing of these objects. To this end, Selection (1) employs a patch simulator (AD-5). The patch simulator generates 64×1360 slot coordinates per second. This rate is constant: if less slots are required, dummy slots are programmed. Only few objects will not obtain a window since a sufficient number of windows per second is available even in worst case crowded skies. These objects are dropped from further windowing and processing. (It is t.b.d. whether the ASM-feature data of such detected but dropped objects shall be transmitted to ground.)

AF1 Data Collection collects the AF1 samples at a rate of

$$64 \times 6 \text{ samples} \times 1360 \text{ Hz} = \quad 522240 \text{ SPS} .$$

It eliminates dummy data, assembles samples to patches, patches to windows, and forwards the window data to subsequent blocks for processing using the GD detection algorithm as before. This procedure is called confirmation. Since AF1 patches are always at least 12 pixels AL, the maximum number of patches which can be collected is given by

$$64 \text{ slots} \times 1360 \text{ Hz} / 12 = 7254 \text{ patches/s}$$

leading to a pixel rate of

$$7254 \text{ patches/s} \times 12 \text{ pix} \times 6 \text{ samples} = 522288 \text{ pix/s}.$$

This pixel rate is nearly the same as before from ASM, so an implementation of the subsequent Detection (1) function is again required in hardware.

As before, Detection (2) is performed in software, and confirmation is obtained by cross-correlation of ASM- and AF1-object features. Confirmation will eliminate all improper objects, and at the final output interface of the AF1 Processing function block, the largest number of stars which can arise is bounded by the maximum number of objects detectable in worst case skies (<1825 obj/s per telescope, i.e. 2x1825=3650 obj/s). However, AF2-BBP5 windowing provides capacity for as many as 7254 objects/s, so this number is presently taken as the upper bound for the data rate at this interface (tbc).

This maximum number of confirmed objects is passed on to the final functional module, responsible for the individual selection for and data collection from each CCD AF2 - BBP5. The output from this module is science data for at most 7254 obj/s. This output is sent to the PDHU.

# 4 ASM PROCESSING

## 4.1 Detection (1)

The main functional modules of detection (1) have been tabulated in Table 4-1. A ROM estimate for the expected gate count is given in the last column of the table. In Table 4-2, these functions are broken down further.

**Table 4-1: Main functional modules of Detection (1)**

| Pos. | Functional Module | ROM estimated logic Kgate count (w/o on-chip memory or FIFO) |
|---|---|---|
| 1 | data interface, distribution of data into three channels (raw data, filtered data, background data) | 5-10 |
| 2 | hyperpixel computation | 50-100 |
| 3 | background pixel interpolation | 50-100 |
| 4 | pixel filtering | 15-30 |
| 5 | pixel SNR computation | 20-40 |
| 6 | extraction of connected pixels | 10-20 |
| 7 | object SNR computation and elimination of noise object pixels | 60-120 |
| **total** | | **210-420 Kgates** |

**Table 4-2: Detailed functional modules of Detection (1)**

| Pos. | Function | ROM estimated gate count (Kgates) |
|---|---|---|
| 1.1 | The data arrives as a pixel stream, line by line. This stream can be forwarded unprocessed for the buffering of the raw data,except that the first and last pixel of every line must be deleted. Copies of the data stream are forwarded to the background module (pos. 2+3) and the filter module (pos. 4). | < 5 |
| 1.2 | Alternatively, for confirmation processing, input data arrives in window format. Special functions are required in order to be able use the detection chip for either case. Presently, the described functions are based only on ASM stream input. | t.b.d. |
| 1.3 | For the background pixel computation, the data stream needs to be collected in 32×32 pixel sets. 30 such pixel sets (which are adjacent in AC) are simultaneously accumulated. The serial input data stream is redirected to the individual sets in turn, thereby avoiding data "corner turn" addressing. | < 5 |
| 2 | Every 32×32 pixel set is processed to yield a single background value (hyperpixel). This involves computing a median twice, as well as calculating one square root per 32×32 set for outlier threshold determination. | |

| | | |
|---|---|---|
| 2.1 | There are several possibilities of hardwiring the computation of the median. The following procedure is one possibility:<br>The 1024 values are buffered serially. The buffered data is read value by value and the most significant bit (MSB = bit no. 16) is evaluated. The data is redirected into two separate serial buffers (FIFO) depending on whether MSB=1 or MSB=0. When this step has been completed, the ones-FIFO is read out, a segment marker is produced, and then the zeros-FIFO is read out. The output buffer to this procedure now contains two separated sections, one with MSB=1 and the other with MSB=0. The content of the output buffer may finally be copied over to the input buffer, overwriting the original data.<br>This procedure is repeated for both sections individually and for bit no. 15. This leads to the data being divided into $\leq$4 sections, sorted w.r.t. the two most significant bits.<br>As this procedure is continued, the number of sections obtained increases until it reaches $\leq$1024 and the data is completely sorted. The median value can now be picked from the sorted data, after the markers have been eliminated.<br>The complete procedure requires at least 16×1024×4 clock cycles or 48 clock cycles per pixel, to be compared with, e.g., 120 clock cycles available per pixel if a 80 MHz clock is used. Several of these median computers will be required in parallel. | 30-60 per median computer |
| 2.2 | A square root requires implementing a processor core which runs the corresponding mathematical approximation program. This is not recommended because this would mean consuming a lot of chip resources for relatively few calls (one per 32×32 pixels i.e. approx. every 1 ms). On the other hand, the range of values required is too large to simply use a look-up-table.<br>The solution proposed is to hardwire an external communication interface with the (Leon) microprocessor in order to delegate this occasional computation to the microprocessor.<br>Alternatively, the thresholding inequality can be squared. This eliminates the need for computing any square roots, but requires a dedicated multiplication and accumulation unit (MAC) because for each pixel value, first the median must be subtracted and the result then be squared. | 20-40 |
| 2.3 | Execute thresholding and redirect data to 2$^{nd}$ median computer. | < 5 |
| 2.4 | Each hyperpixel is appended to a background queue (=stripmap of hyperpixels) which is 30 pixels AC. | - |
| 3.1 | Whenever a line in the background queue has been completed, this line is extra- and interpolated (in 1-d) to yield 981 background values.<br>Interpolation is linear, so it can be formulated recursively to look like this: | 20-40 |

| | | |
|---|---|---|
| | $y_0, y_{32}$ : any two adjacent hyperpixels<br>$y_k, k = 1,...,31$ : desired interpolation pixels<br>$y_k = y_{k-1} + \alpha$ where $\alpha = \frac{y_{32} - y_0}{32}$<br>To execute interpolation, an integer addition unit is required. Division by 32 can be performed by bit shift. | |
| 3.2 | Extrapolation is by mirror-symmetric reflection of interpolated values. Addressing is required to fetch the proper data and to align the computed 981 background values in correct order.<br>Each background value is appended to a background queue (with pixels still missing along-scan). | 10-20 |
| 3.3 | Whenever a new background pixel has been computed, the 31 intermediate values between this pixel and the corresponding former pixel at same AC-position are determined by means of the same linear interpolation as before. | 20-40 |
| 3.4 | The background data is aligned as a line-oriented stream for synchronous association with the filtered pixel stream. | < 5 |
| 4.1 | On the filter channel, nine copies of the pixel stream are generated and respectively delayed such that at each strike of the clock, nine registers provide the neighboring values required by the 2-d filter. The filter coefficient multiplication is performed simply by bit shifts of register contents. An integer adding unit (24 bits) is required which sums the shifted contents of the nine registers. Division of the sum is again performed by bit shift. | 10-20 |
| 4.2 | Special treatment is required at the two boundaries of the stripmap (or at the boundaries of the confirmation window). | < 5 |
| 4.3 | The lines of buffered raw data, background data, and filtered data are re-synchronized to match the same pixel position. The data is still a line-oriented, serial stream, with a line width of 981. | < 5 |
| 5.1 | For every pixel, the pixel-SNR is computed:<br>$SNR_{ij}^2 = \left(p_{ij}^{filtered} - p_{ij}^{background}\right)^2 \cdot \left(p_{ij}^{filtered} + RON^2\right)^{-1}$<br>This requires a 32 bit, fixed point MAC and a look-up table for inversion. The look-up table has 2**16=65536 entries and may have to reside in chip-external memory.<br>It is also required to compute the sign<br>$\mathrm{sgn}_{ij} = \mathrm{sgn}\left(p_{ij}^{filtered} - p_{ij}^{background}\right)$<br>The two values $SNR_{ij}^2$ and $\mathrm{sgn}_{ij}$ are added to the pixel stream for subsequent use. | 20-40 |

| | | |
|---|---|---|
| 5.2 | Pixels are thresholded by evaluation of the equation<br>$SNR_{ij}^2 > t_1$ ?<br>In correspondence to the pixel stream, a binary data stream is generated which reflects the outcome of the thresholding. | - |
| 6.1 | A connectivity flag is attributed to those pixels of the binary stream that assume the value 1 and whose 8 immediate neighbors have the same property. This is implemented by generating eight copies of the binary data stream and delaying them such that they appear synchronized in nine registers which are ANDed. The resulting binary stream assumes the value 1 at every pixel where connectivity is confirmed, i.e. at every pixel which is member of an object. | 5-10 |
| 6.2 | Special treatment is required at the two boundaries of the stripmap (or at the boundaries of the confirmation window). | < 5 |
| 7 | From these (OM=object member) pixels, it is required to extract those pixels which are members of true objects (as opposed to noise objects), i.e. objects with sufficient overall SNR². It is not attempted to hardwire object-based functions; processing at object level shall be done in software. However, in order to reduce as much as possible the data rate at the output of the hardwired function, only those pixels which belong to true objects shall be output. Although this requires the computation of object SNR², it does not require the handling of objects as data entities. | |
| 7.1 | In each line of data, 1-d connected sets of OM-pixels are each assigned a unique label *a* (integer value $a \geq 1$). These labels are entries in a look-up-table with function values *X(a)* and *F(a)*. For labels which are in use *X(a)=1*, for labels which are still (or again) available *X(a)=0*. For each 1-d connected set, the value<br>*F(a)=sum of signed SNR² of all pixels belonging to set a*<br>is computed and stored in the table.<br>Pixels *y* which are members of object *a* are labelled *A(y)=a*. Pixels *y* which are not OM-pixels obtain the label *A(y)=0*.<br>The function *A(y)* is maintained for a t.b.d. (e.g. 20) number of lines. | 20-40 |
| 7.2 | The connectivity of sets from the current line with sets from the preceding line is identified. Set label equivalencies are established. (*n* labels are equivalent if each labels part of the same 2-d connected set.) | 20-40 |
| 7.3 | Equivalent labels are overwritten using the lowest involved label *â* of the current line. The pixel label function *A(y)* is updated accordingly for the current line. The object SNR² *F(â)* associated with a replacement label *â* is the sum of the signed SNR² from all equivalent labels. Equivalent labels *a* which have been overwritten are set free, and the SNR value *F(a)* is deleted. | 20-40 |

| | | |
|---|---|---|
| 7.4 | OM-pixels are output, however for a line which is t.b.d. (e.g. 20) lines older than the current line. A pixel *y* is output only if it is a member of an object with sufficient SNR, i.e. if<br>$A(y)>0$ and $F(A(y))>t_2$<br>where $t_2$ is the object SNR² threshold. This condition is evaluated using the look-up-table. | 5-10 |

## 4.2 Detection (2)

Table 4-3 exhibits the subfunctions of Detection (2). These functions are essentially object-based and significantly more complex than the pixel-based ones, involving considerable addressing and

numerics. Their execution requires a microprocessor architecture, so software implementation on an existing microprocessor is recommended. A reliable theoretical estimation of the associated processor load does not seem feasible. This load should therefore be determined by prototyping the functions on an appropriate trial processor.

Nevertheless, a ROM-assessment of the expected processing load in three simple categories *low, medium, high* has been introduced into the table. The processing load is assessed "medium" if a small number of operations (e.g. 1 fetch, 1 multiply/accumulate, 1 address, 1 store) is required for every pixel of the object; it is assessed "high" if more than ~5 operations are required for every pixel of the object; and finally, it is assessed "low" if not every pixel is involved in processing the respective function, and if this function requires few operations. **A typical object is assumed to consist of 20 pixels, and a typical operation is assumed to require 5 instructions.**

**Table 4-3: SW-subfunctions of Detection (2)**

| Pos. | Function | Estimated processor load per object |
|---|---|---|
| *object-based functions:* | | |
| 1 | generate, update, close object buffers<br>(receive object-member pixel data; assign object buffer; update object buffer with received data; close object buffer when current line supplies no further OM-pixels) | medium |
| 2 | execute watershed-based object decomposition[1] algorithm, update object buffer | high |
| 3 | compute object max and bounding box | medium |
| 4 | compute background flux | medium |
| 5 | compute object flux | medium |
| 6 | compute object barycenter, shift in AL and AC | medium |
| 7 | perform $\chi^2$-test on object, refine barycenter estimate (AL) and flux estimate (if $\chi^2$-test is successful) | high |
| 8 | compute 2nd order moments, determine eccentricity and orientation | high |
| 9 | determine object type | low |
| 10 | correct magnitude for bright stars based on spline interpolation | low |
| 11 | compute predicted AF1 $\xi$-coordinate (AL) of object center | low |
| 12 | assign ID to each object in order of increasing AF1 $\xi$-coordinate | low |
| 13 | output object ID to next processing stage | low |

[1] In the terminlogy of AD-8 , "object decomposition" is called "detection deblending".

# 5 AF1 PROCESSING

## 5.1 AF1 Selection (1)

This software function receives all detected objects from both the ASM1 and ASM2 channels and assigns non-conflicting patches for AF1-windowing. Outputs from this function are

1. AF1 slot parameters, to be forwarded to FPA
2. look-up-table (LUT2) entries for later retrieving an object address from any particular AF1 line and slot
3. t.b.d.: science data from detected objects which have been deselected

Table 5-1 exhibits a breakdown of this function into modules, again including an indication of the expected processor load in three categories.

**Table 5-1: Functional modules of AF1 Selection (1)**

| Pos. | Function | Estimated processor load per object/ per line |
|---|---|---|
| *object-based functions:* | | |
| 1 | reception of object ID from both detection outputs | low |
| 2 | assignment of a unique address to each object from either telescope (maximum of 64 per AF1 line) | low |
| 3 | compute predicted AF1 $\eta$-coordinate (=AC) of object center | low |
| 4 | look-up patch size(s) and determine ideal patch coordinates for AF1 windowing (depending on object type) | low |
| 5 | enter patches (of typical size $12\times6$) into patch simulator; apply collision strategy (shift/ merge/ drop) | medium-high |
| *line-based functions:* | | |
| 6 | determine and output slot coordinates for current AF1 line; store associated object addresses in LUT2 for recovery after read-out | $64\times10=640$ instr. |
| 7 | "scroll" patch simulator to next AF1 line | negl. |

## 5.2 AF1 Data Collection

The samples obtained linewise by AF1 read-out are assembled to patches and windows by this function. Each window is associated with a unique object address by referring to LUT2. Window data is passed on to a hardwired module for detection (1) processing.

**Table 5-2: Functional modules of AF1 Data collection**

| Pos. | Function for each CCD | Estimated processor load per window (instr.) |
|---|---|---|
| *patch/window-based functions:* | | |
| 1 | receive and buffer slot data in 64 simultaneous patch buffers; assemble samples to 12×6 patches | 12×6×5 =360 |
| 2 | look up associated object address in LUT2 | 10 |
| 3 | assemble patches to windows | negl. |
| 4 | output window data to hardwired Detection (1) function | 12×6×5=360 |
| 5 | perform numerical binning of transmission window | 12×6×5=360 |
| 6 | look-up datation code | 10 |
| 7 | prepare ASM transmit window | < 100 |
| 8 | packetize data for output to PDHU | tbd |
| | total | ~1200 |

## 5.3 AF1 Detection (1)

Confirmation processing requires applying the GDA to the window data obtained from AF1. Only bright stars are exempt from this requirement (t.b.c.). The window data obtained from the last functional block therefore undergoes the Detection (1) stage as before (cf. section 4). It has already been required that the respective chip shall be implemented such that it accepts pixel input either in stripmap or window format.

The output from this functional block are the object-member-pixels detected in each window. This collection of OM-pixels is passed on for further detection processing (Detection (2)) and object confirmation in the subsequent software module.

Due to the pixel input rate, the worst case processing load for confirmation detection (1) processing is close to that of one ASM chain.

## 5.4 AF1 Detection (2) and Selection (2)

AF1 Detection (2) is identical to the software function described in section 4.2. As a result of this processing stage, with each window there are associated one or several objects plus their features.

AF1 selection (2) consists of cross-correlating these feature sets with those originally obtained from ASM detection (2). The results of this cross-correlation are used to either confirm or discard objects. Confirmed objects are used to compute scan rates. If the scan rate is beyond a threshold, the object is classified as moving, and for this object, further tracking is based on the computed individual scan rates instead of the current AOCS-based scan rates.

The processing load for this software module is twice that of section 4.2, as objects from both telescopes have to be processed.

**Table 5-3: Functional modules of Detection (2) and Selection (2)**

| Pos. | Function for each CCD | Estimated processor load per object |
|---|---|---|
| 1-13 | cf. Table 4-3 | |
| 14 | retrieve ASM features from LUT1 | low |
| 15 | cross-correlate AF1 features with ASM features for each object | medium |
| 16 | compute AL- and AC-deviation relative to positions predicted by average scan rates | low |
| 17 | identify moving objects; compute individual scan rates | low |

# 6 AF2-BBP5 PROCESSING

## 6.1 AF2-BBP5 Selection

AF2-BBP5 Selection is very similar to AF1 Selection (1). The associated maximum processing load is therefore estimated at 15 times the figure for AF1 Selection (1).

## 6.2 AF2-BBP5 Data Collection

AF2-BBP5 data collection is similar to AF1 data collection but requires less instructions due to FPA-binning in AC. Also, in the worst case window rate, patches are only 6 samples long (AL).

**Table 6-1: Functional modules of AF1 Data collection**

| Pos. | Function for each CCD | Estimated processor load per window (instr.) |
|---|---|---|
| *patch/window-based functions:* | | |
| 1 | receive and buffer slot data in 32(64) simultaneous patch buffers; assemble samples to patches | 6×5 =30 |
| 2 | look up associated object address in LUT3 | 10 |
| 3 | assemble patches to windows | negl. |
| 4 | output window data to hardwired Detection (1) function | 6×5=30 |
| 5 | perform numerical binning of transmission window | tbd (only AF11 and BBP1-5) |
| 6 | look-up datation code | 10 |
| 7 | packetize data for output to PDHU | tbd |
| | total | ~100 |
| | | |

# 7 PROCESSING LOAD ANALYSIS

## 7.1 Hardware

An ASIC implementation of Detection (1) has been estimated at 210-420 Kgates (excluding on-chip memory requirements). This number is to be compared e.g. with ~1.2 Mgates available design complexity announced by Atmel for the forthcoming, largest MH1RT ASIC matrix. This means that at least two of the three required Detection (1) modules may be placed on one chip. Two identical modules are required for ASM stripmap processing, and a third one will have to be adapted for AF1 input data in window format. It is thus proposed to develop one Detection (1) chip (with 2 modules) for ASM stripmap processing, and another, similar one for AF1 window processing.

There is no doubt that the real-time performance of the chips will be sufficient for the present application.

The particuliarity of the chips stems from the diversity of the functions to be implemented. Normally, large ASIC possess a substantial number of identical or similar modules which run in parallel. In this case, there are as many as 7 totally different, complex functions to be implemented.

## 7.2 Software

### 7.2.1 General

A rough order of magnitude (ROM) of the required total processing load (for one ASTRO-VPU, i.e. one row of CCD) shall be obtained by counting instructions (Table 7-1). From each of the function blocks analyzed in the preceding sections, the number of subfunctions with low, medium, and high complexity is counted, respectively. For the number of instructions associated with these degrees of complexity, see the introductory remark in section 4.2.

The right-most column of the table exhibits the ROM of MIPS required for the individual function blocks. If this is compared to an expected processing performance of 40-80 MIPS for the Leon microprocessor, the following conclusions can be drawn.

One or two microprocessors each are required for the two Detection (2) functions, corresponding to ASM (original) Detection (2) and AF1 (confirmation) Detection (2).

Selection (1) leads to a very high processing load. This is due on one hand to the patch simulator, which in principle is based on computationally expensive pixel filling (cf. section 7.2.2). On the other hand, such a simulator is required for each of 16 CCD.

An additonal microprocessor will be required to perform the remaining 3 processing functions.

**Table 7-1: Processing load estimate for software functions**

| function block | instructions | | | applicable rate | MIPS |
|---|---|---|---|---|---|
| | low (20 per call) | medium (20×20 per call) | high (20×100 per call) | | |
| detection (2) Table 4-3 | 5×20 | 5×400 | 3×2000 | 7300 objects/sec for both ASM1 and ASM2 | 59.2 |
| selection (1) (object-based) Table 5-1 | 4×20 | 1×1000 | 0 | 16×7254 objects/sec for all CCD | 125.4 |
| selection (1) (line-based) Table 5-1 | 6×640 (AF1 & BBP1-5) 10×320 (AF2-AF11) | | | 1360 lines/sec | 9.6 |
| AF1 data collection Table 5-2 | 1200 | | | 7254 windows/sec | 8.7 |
| AF2-BBP5 data collection Table 5-2 | 100 | | | 15×7254 windows/sec for AF2-BBP5 | 10.9 |
| AF1 detection (2) and selection (2) Table 5-3 | 8×20 | 6×400 | 3×2000 | 7254 objects/sec | 62.1 |
| **total** | | | | | 275.9 |

### 7.2.2 Selection (1)

It is clear that the processing load for Selection (1) (object-based) must be reduced dramatically. This load is essentially due to the patch simulator. The present load estimate is based on the assumption that a patch is entered into the simulator by marking each pixel covered by the patch. This leads to a large number of individual addressing steps, because a patch typically consists of 6×12=72 pixels. If a single address calculation, data fetch and data put step requires e.g. 10 instructions, already the following product grows rather large:

$$10 \text{ instructions} \times 72 \text{ pixels} \times 7254 \text{ patches/sec} \times 16 \text{ CCD} \approx 84 \times 10^6 \text{ instr./sec}$$

In the actual implementation, additional instructions will arise because 2-d addressing is required, patches have to be shifted or merged, etc.

A solution to this problem will be either a smarter algorithm or a hardwiring solution. The latter in its core would be an architecture which repeatedly calculates a pixel address within a given patch, reads a memory structure at that address, processes the pixel value (identifying whether the pixel is void or occupied), and returns the processed value to memory. This architecture can be made lean because the pixels value can be respresented by just 1 bit and the address range is rather limited. Presumably 16 copies of the same function could be placed on one chip for simulaneous execution.

# 8 CONCLUSION

The present technical note assesses the requirements associated with a one-to-one implementation of GDA and the associated data processing and programming for each row of 18 ASTRO CCD.

Due to the large number of completely diverse functions many of which are data dependent and require numerical or recursive algorithms, an end-to-end software solution would seem appropriate but requires too much processor power.

It is therefore required to hardwire computationally intensive functions which are not so complex that their hardwiring would result in reproducing a microprocessor architecture with fixed program. This approach has been taken presently and has shown that the pixel-based detection stages of GDA and perhaps also the patch simulator should be hardwired. This leads to **three different ASIC**, two similar ones (for detection based on stripmap and window data, respectively) and another, completely different one for patch simulation. The first two ASIC are internally very diverse (comprising 7 totally different functions), while the latter would have 16-fold internal redundancy.

The remaining software functions have been estimated to require around 150 MIPS, assuming that a typical operation (address computation, data fetch/return, multiply/accumulate etc.) requires 5 instructions. The baseline microprocessor is the future Leon which is characterized by 120 MIPS max, 40 MIPS average processor performance. It is presently **not clear how many Leon processors** would actually be required to produce this performance, particularly if the efficiency of existing compilers is taken into account. A **clarification of this issue requires a breadboarding exercise** which is beyond the scope of phase 1 of this study. Also the **150 MIPS is just a ROM estimate** which may prove incorrect if code is actually prototyped.

For comparison, a single commercial microprocessor with typically 1 GIPS performance would be sufficient for an end-to-end software implementation.

It should finally be noted that the high implementation requirements arising in this analysis are due to the satisfaction of several maximum requirements:

- adaptation of real-time processing performance to worst case sky densities
- ASM detection with 100% overhead for false objects (cosmic hits)
- AF1 windowing using 64 slots which represents a 100% overhead for false objects
- AF2-BBP5 slot selection and data collection at maximum theoretical patch placement of $32 \times 1360/6 = 7254$ patches/sec which represents a 100% margin over worst case sky densities