

Cray XD1

Mini-curso de Computação Híbrida Reconfigurável*

Vitor C. F. Gomes, Andrea S. Charão, Haroldo F. C. Velho

Última atualização: 19 de setembro de 2009

Universidade Federal de Santa Maria / Instituto Nacional de Pesquisas Espaciais
Cray XD1 / Mini-curso de Computação Híbrida Reconfigurável

1 Cray XD1

O Cray XD1 é um sistema híbrido reconfigurável de alto desempenho lançado em outubro de 2004. Trouxe no momento de seu lançamento algumas inovações tecnológicas. Entre elas está a rede de interconexão de alto desempenho RapidArray, otimizações no sistema Linux e a inclusão de FPGAs em seu chassis. Na figura 1 é possível ver um *rack* com diversos equipamentos XD1 e um equipamento em destaque.



Figura 1: Cray XD1. Fonte: [Cray Inc. 2005a]

*Este mini-curso é uma ação vinculada ao convênio INPE/UFSM e tem por objetivo oferecer uma introdução a Computação Híbrida Reconfigurável

1.1 Arquitetura

Cada sistema Cray XD1 é composto por seis nós (*blades*), cada um contendo dois processadores de propósito geral AMD Opteron 64bits e um FPGA Xilinx Virtex II Pro. A arquitetura de um *blade* do Cray XD1 pode ser vista na figura 2. É possível observar que o dispositivo reconfigurável tem acesso direto a quatro bancos de memória QDR II SRAM, que possuem 4MB cada. O RapidArray Processor permite que os processadores enviem dados para o FPGA e que o FPGA leia dados da DRAM. No desenvolvimento de aplicações híbridas para este sistema, existem duas questões chaves que devem ser observadas: a transferência de dados entre os dispositivos e o uso eficiente dos diferentes níveis de memória disponíveis no sistema.

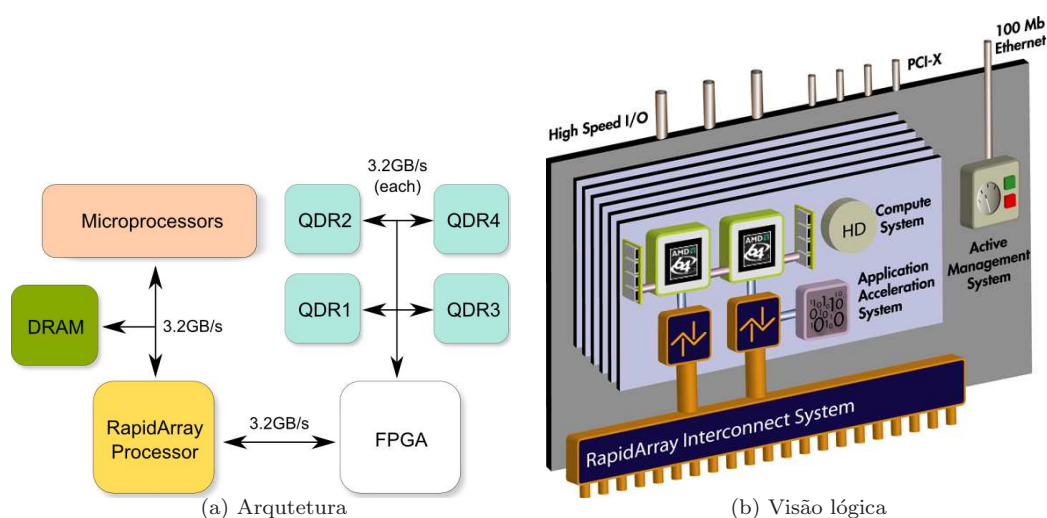


Figura 2: Blade do Cray XD1

1.2 Hierarquia de Memória

Conforme pode ser visto na Figura *fig:diagrama-blade*, o FPGA de um *blade* do XD1 tem acesso a diferentes tipos de memória. Além dos quatro blocos de SRAM e da memória SRAM, o FPGA pode utilizar blocos de memória internos ao hardware reconfigurável.

O uso destes três níveis de memória deve ser planejado no desenvolvimento de uma aplicação, para evitar a redução de eficiência pelo gargalo no acesso a memória. Cada recurso disponível tem forma de acesso e quantidades distintas. A DRAM é o mais alto nível, com a maior quantidade de memória disponível e com latência de leitura não constante. Esta memória pode ser acessada usando uma interface de comunicação disponibilizada pela Cray, o *RapidArray Transport Core*. Em um nível mais baixo, estão os bancos de memória QDR II SRAM com 4MB cada. A latência de acesso a esses bancos é de 8 ciclos para a leitura, e o acesso é feito utilizando o *QDR II SRAM Core*, também disponibilizado pelo fabricante. Além desses, a família de FPGAs Vitex Pro possuem bancos de memória internos que podem ser acessados diretamente em um único ciclo.

Normalmente a DRAM é utilizada para compartilhar dados com o FPGA, sendo carregados para a QDR II RAM para serem acessados durante a execução da aplicação. A memória interna do FPGA, disponível em menor quantidade, é utilizada para registros e cache de dados. Independente de uma forma geral,

cada problema necessitará de recursos e formas de acesso diferentes que devem ser otimizadas conforme a necessidade.

1.3 Comunicação entre CPU e FPGA

A comunicação entre CPU e FPGA é um elemento chave que deve ser levado em consideração durante o desenvolvimento de aplicações nesta tecnologia.

Para a comunicação entre dispositivos em um *blade*, a Cray disponibiliza a API RapidArray Transport Core que é um componente que deve ser usado na descrição do algoritmo em VHDL. Esta entidade é composta por dois blocos denominados Fabric Request e User Request [Cray Inc. 2005f]. Estes blocos permitem duas formas de comunicação que serão vistas nas subseções a seguir.

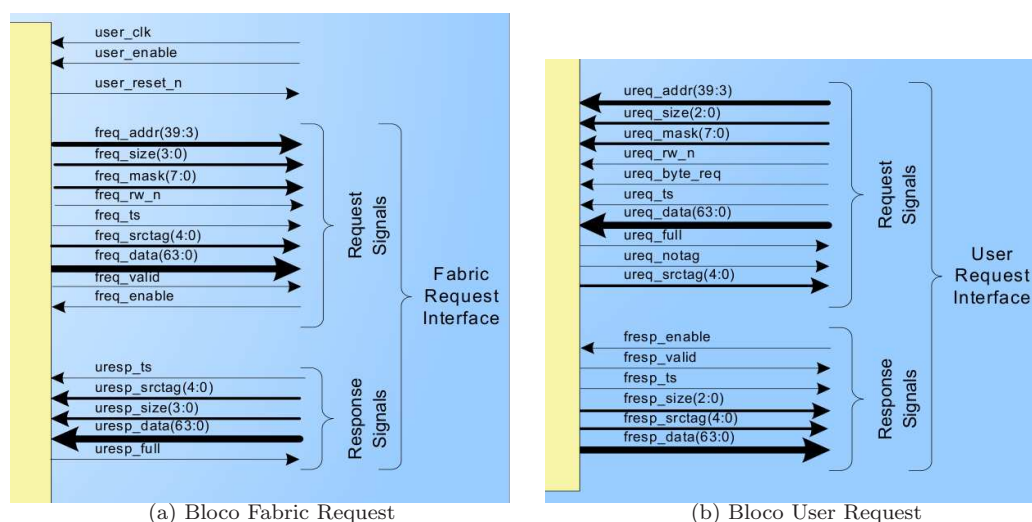


Figura 3: Blocos do RapidArray Transporte Core

1.3.1 Bloco Fabric Request

O bloco Fabric Request, que realiza a comunicação usando uma abordagem *push*, permite que o programa executado nos processadores envie e requisite dados para o FPGA. Esta abordagem mantém os processadores ocupados durante a transferência de dados. Para permitir esta comunicação, a Cray disponibiliza a biblioteca *enlib*, a qual abstrai ao programa o FPGA como um arquivo, deste modo, a transferência de dados entre os processadores e o FPGA é realizada através de leituras e gravações pelo programa em C neste 'arquivo'. Com a realização de uma leitura ou escrita, o FPGA recebe, através do RapidArray Transport Core, uma requisição que deve ser tratada pela aplicação do FPGA. Em caso de leitura, deverá ser retornado um valor ao RapidArray Transport Core para que ocorra o retorno da função chamada pelo programa em linguagem de alto nível. Somente é permitida a manipulação de um *quadword* (64 bits) por requisição utilizando o bloco Fabric Request [Cray Inc. 2005f]. Na Figura 3a é possível ver os sinais que são ativados quando uma requisição é feita pelo programa (*Request Signals*) e os sinais que são usados pela aplicação em VHDL enviar informações para o programa (*Response Signals*).

Um trecho de programa que utiliza a biblioteca *einlib* para acessar o FPGA pode ser visto na Listagem 1. Nele é possível ver a abertura e o fechamento do FPGA como um arquivo, o carregamento do arquivo de configuração no FPGA e a gravação e escrita de um dado no FPGA. Apesar do programa em C, considerar o FPGA como um arquivo, que pode ser acessado em posições aleatórias, o que é visto no lado do FPGA é diferente. Após solicitar uma gravação no FPGA via programa (linha 11), através do *RapidArray Transport*, o sinal *s_freq_ts* do bloco *Fabric Request* sinaliza a aplicação do FPGA que existe um dado válido no barramento a ser lido. O endereço informado no programa é repassado ao FPGA via *freq_addr*, permitindo que a aplicação trate desta requisição da maneira que for conveniente.

Para a resposta do FPGA ao programa os dados devem ser colocados no barramento *uresp_data* e o sinal *uresp_ts* deve ser alterado para nível lógico alto¹. Através de envios e leituras feitos pelo programa ao FPGA é possível transferir os dados necessários para a computação da aplicação no hardware reconfigurável. Entretanto, para grandes quantidade de dados – maior que 16KB – é aconselhado que a comunicação seja feita via *User Request*, pois apresenta melhores taxas de transferência.

Listagem 1: Uso da *einlib* para acessar o FPGA

<code>#include <stdio.h></code>	1
<code>#include "einlib.h"</code>	2
<code>...</code>	3
<code>uint64_t read_data;</code>	4
<code>uint64_t write_data = 1;</code>	5
<code>// abre o 'arquivo' FPGA</code>	6
<code>int fp_fd = fpga_open("/dev/ufp0", ORDWR O_SYNC, &erro);</code>	7
<code>// carrega o arquivo de configuracao</code>	8
<code>fpga_load(fp_fd, "arquivo.bin.upf", &erro);</code>	9
<code>// grava 1 na posicao 0x08</code>	10
<code>status = fpga_wrt_appif_val(fp_fd, write_data, 0x08, TYPE_VAL, &erro);</code>	11
<code>// le os dados da posicao 0x08</code>	12
<code>status = fpga_rd_appif_val(fp_fd, &read_data, 0x08, &erro);</code>	13
<code>// fecha o 'arquivo' FPGA</code>	14
<code>fpga_close(fp_fd, &erro);</code>	15
<code>...</code>	16

1.3.2 Bloco User Request

A abordagem que utiliza o bloco User Request é conhecida como *pull* e, diferentemente da Fabric Request, mantém os processadores livres durante a transferência de dados entre o programa em C e o FPGA. Para isso, este bloco permite que o FPGA realize leituras e escritas em um espaço de memória compartilhado do programa. O endereço para a região de memória, que é compartilhada utilizando a *einlib*, é enviado ao FPGA através do bloco Fabric Request. Estando disponível o endereço, a aplicação descrita para o FPGA é capaz de fazer até 32 requisições sequenciais ao RapidArray Transport Core. Cada requisição pode solicitar até 8 posições contíguas da memória do programa. Os retornos das solicitações ao RapidArray Transporte Core não são necessariamente na ordem em que foram realizadas. O *core* garante somente a ordem das 8 posições contíguas de cada requisição [Cray Inc. 2005f]. O sistema descrito para o FPGA deve ordenar os

¹Para a comunicação são utilizados todos os sinais do bloco. Os demais sinais são omitidos para facilitar a compreensão do funcionamento da comunicação, sendo apresentados os principais. Mais informações podem ser encontradas em [Cray Inc. 2005f]

dados através do auxílio de *tags* disponibilizadas durante a requisição e o retorno. Outra solução é aguardar o retorno de uma solicitação antes de realizar outra.

Os sinais do bloco *User Request* podem ser vistos na Figura 3b. Através dos sinais *Request Signals* o FPGA é capaz de requisitar posições de memória, sendo atendidos através dos sinais *Response Signals*. Um programa que compartilha uma região de memória e envia seu endereço para o FPGA pode ser visto na Listagem 2. Após o recebimento do endereço, o FPGA está apto a realizar requisições de posições de memória através dos sinais *ureq_addr* e *ureq_ts* do bloco *User Request*. Os dados serão recebidos através dos sinais *fresp_ts* e *fresp_data*².

Listagem 2: Compartilhamento de memória com o FPGA

```
#include <stdio.h>
#include "einlib.h"
...
volatile uint64_t *buf_ptr;
// abre o 'arquivo' FPGA
int fp_fd = fpga_open("/dev/ufp0", ORDWR|O_SYNC, &erro);
// carrega o arquivo de configuracao
fpga_load(fp_fd, "arquivo.bin.upf", &erro);
// aloca regioao para compartilhamento
buf_ptr = (volatile uint64_t*) fpga_set_ftrmem (fp_fd, 9, &erro);
// atribui valores na regioao compartilhada
for(i=0; i<n; i++){
    buf_ptr[i] = (uint64_t)i;
}
// envia o endereco para o FPGA
status = fpga_wrt_appif_val (fp_fd, (uint64_t)buf_ptr, 0x00, TYPE_ADDR, &erro);
...
// fecha o 'arquivo' FPGA
fpga_close (fp_fd, &erro);
...
```

1.4 Uso do RapidArray Transport Core: *rt_core*

A Cray disponibiliza um *template* com a estrutura de um projeto para o desenvolvimento de aplicações híbridas para o Cray XD1. Este *template* possui um componente inicial que engloba os sinais de saída e entrada no FPGA (*reset_n*, *clock*, etc), uma instância do componente *rt_core*, o qual permite a comunicação com o programa executado em CPU e uma instância do *qdr2_core*, que é a interface para acessar os blocos de memória DRAM. Uma ilustração do arquivo VHDL que fornece esta estrutura pode ser vista na Figura 4. A aplicação desenvolvida é representada pela quadrado amarelo *user_app*. Os demais componentes estão previamente configurados no *template*, devendo a aplicação tratar os sinais de comunicação, quando necessário. Na Listagem 3 é mostrado uma máquina de estados que faz manipulação das requisições feitas pelos programas escritos em alto nível mostrados nas Listagens 1 e 2.

Listagem 3: Fazer

²Para a comunicação são utilizados todos os sinais do bloco. Os demais sinais são omitidos para facilitar a compreensão do funcionamento da comunicação, sendo apresentados os principais. Mais informações podem ser encontradas em [Cray Inc. 2005f]

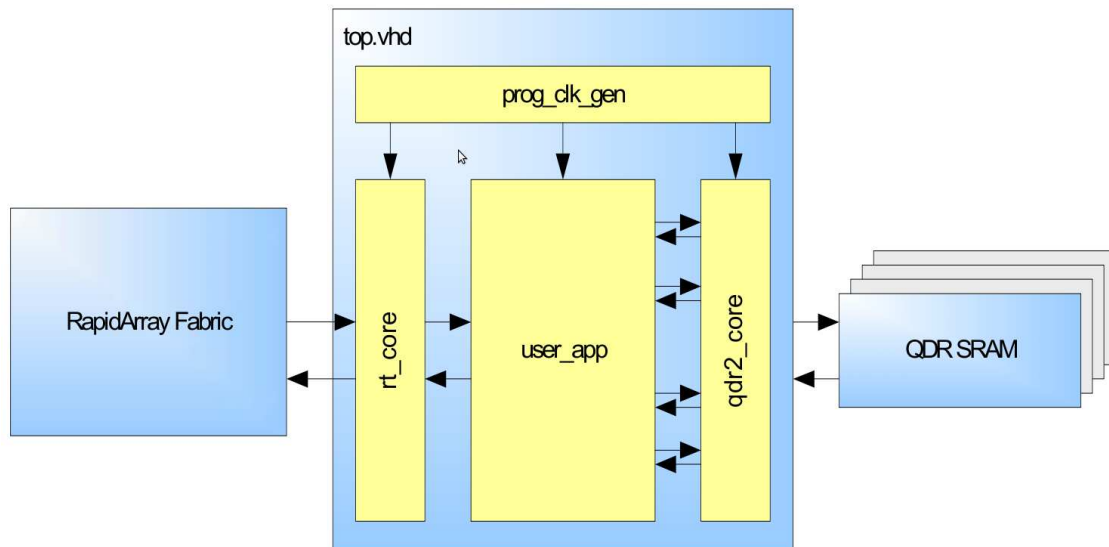


Figura 4: Estrutura para desenvolvimento de aplicação para FPGA no Cray XD1. Fonte: [Cray Inc. 2005e]

```

architecture basic of comunicacao is
  — Sinais de controle e registradores
  type t_req_state is (idle, read, read_blocked, write);
  signal s_req_state: t_req_state;
  signal s_buf_ptr : std_logic_vector(39 downto 3);
  signal s_data : std_logic_vector(63 downto 0);
  signal s_state_enable: std_logic;
  — Sinais do bloco Fabric Request
  signal s_freq_addr      : std_logic_vector(39 downto 3); — req address
  signal s_freq_size     : std_logic_vector(3  downto 0); — req size
  signal s_freq_mask    : std_logic_vector(7  downto 0); — req byte mask
  signal s_freq_rw_n    : std_logic; — req read/write
  signal s_freq_ts      : std_logic; — req transfer start
  signal s_freq_valid   : std_logic; — req valid
  signal s_freq_data    : std_logic_vector(63 downto 0); — req write data
  signal s_freq_srctag  : std_logic_vector(4  downto 0);
  signal s_freq_enable  : std_logic; — s_enable req interface
  signal s_uresp_full   : std_logic; — resp buffer full
  signal s_uresp_ts     : std_logic; — resp transfer start
  signal s_uresp_size   : std_logic_vector(3  downto 0); — resp size
  signal s_uresp_data   : std_logic_vector(63 downto 0); — resp data
  signal s_uresp_srctag : std_logic_vector(4  downto 0); — resp source tag
begin

  freq_reg : process(user_clk, reset_n) is
  begin — process freq_reg
    if (reset_n = '0') then
      s_state_enable <= '0';
    elsif (user_clk 'event and user_clk = '1') then

```

```

— armazena a requisicao
if (s_freq_enable = '1') then
    s_freq_ts      <= freq_ts;
    s_freq_size   <= freq_size;
    s_freq_srctag <= freq_srctag;
    s_freq_addr   <= freq_addr;
    s_freq_rw_n   <= freq_rw_n;
    s_freq_valid  <= freq_valid;
    s_freq_mask   <= freq_mask;
    s_freq_data   <= freq_data;
end if;
— sinal de habilitacao da maquina de estados
s_state_enable <= user_enable and rt_ready and qdr_ready;
end if;
end process freq_reg;

access_control : process(user_clk , reset_n) is
begin
    if (reset_n = '0') then
        s_dados      <= (others => '0');
        s_buf_ptr    <= (others => '0');
        s_freq_enable <= '0';
        s_req_state  <= idle;
    elsif (user_clk'event and user_clk = '1') then
        s_uresp_full <= uresp_full;
        s_uresp_ts   <= '0';
        if(s_state_enable = '1') then
            case (s_req_state) is
                when idle =>
                    if (s_freq_ts = '1' and s_freq_valid = '1') then — dado valido
                        if (s_freq_rw_n = '1') then — leitura
                            if (s_uresp_full = '0') then — buffer vazio
                                s_req_state <= read;
                            else
                                s_req_state <= read_blocked;
                            end if;
                        else — escrita
                            s_req_state <= write;
                        end if;
                    else — dado invalido
                        — mantem em idle , e mantem habilitada novas requisicoes
                        s_req_state <= idle;
                        s_freq_enable <= '1';
                    end if;
                when read =>
                    if (s_uresp_full = '0') then — buffer de envio vazio
                        case (s_freq_addr(6 downto 3)) is
                            when "0000" => — 0x00UL => Ponteiro
                                s_uresp_data <= "000" & x"000000" & s_buf_ptr;
                            when "0001" => — 0x08UL => Dados

```

```

        s_uresp_data <= s_data;
        when others => null;
    end case;
    — envia o registrador requisitado
    s_uresp_ts    <= '1'; —habilita envio
    s_uresp_size  <= "0000";
    s_uresp_srctag <= s_freq_srctag;
    — habilita novas requisicoes e vai para estado idle
    s_req_state   <= idle;
    s_freq_enable <= '1';
    else — buffer de envio cheio
        s_req_state <= read_blocked;
    end if;
when write =>
    case (s_freq_addr(6 downto 3)) is
        when "0000" => — 0x00UL => Ponteiro
            if (s_freq_mask(0) = '1') then
                s_buf_ptr(7 downto 3) <= s_freq_data(7 downto 3);
            end if;
            if (s_freq_mask(1) = '1') then
                s_buf_ptr(15 downto 8) <= s_freq_data(15 downto 8);
            end if;
            if (s_freq_mask(2) = '1') then
                s_buf_ptr(23 downto 16) <= s_freq_data(23 downto 16);
            end if;
            if (s_freq_mask(3) = '1') then
                s_buf_ptr(31 downto 24) <= s_freq_data(31 downto 24);
            end if;
            if (s_freq_mask(4) = '1') then
                s_buf_ptr(39 downto 32) <= s_freq_data(39 downto 32);
            end if;
        when "0001" => — 0x08UL => Data
            s_data <= s_freq_data;
        when others => null;
    end case;
    — habilita novas requisicoes e vai para idle
    s_req_state   <= idle;
    s_freq_enable <= '1';
when read_blocked =>
    if (s_uresp_full = '0') then
        s_req_state <= read;
    else
        s_req_state <= read_blocked;
    end if;
when others => null;
end case;
end if;
end if;
end process access_control;
end architecture basic;

```

Referências

- [Chamberlain et al. 2008] Chamberlain, R. D., Lancaster, J. M., and Cytron, R. K. (2008). Visions for application development on hybrid computing systems. *Parallel Comput.*, 34(4-5):201–216.
- [Cray Inc. 2005a] Cray Inc. (2005a). *Cray XD1 Datasheet*. Mendota, MN, USA.
- [Cray Inc. 2005b] Cray Inc. (2005b). Cray XD1 FPGA Development. Mendota Heights, MN, USA. Cray Inc.
- [Cray Inc. 2005c] Cray Inc. (2005c). *Cray XD1 FPGA Programming*. Mendota, MN, USA.
- [Cray Inc. 2005d] Cray Inc. (2005d). *Cray XD1 System Overview*. Mendota Heights, MN, USA.
- [Cray Inc. 2005e] Cray Inc. (2005e). *Design of Cray XD1 QDR II SRAM Core*. Mendota, MN, USA.
- [Cray Inc. 2005f] Cray Inc. (2005f). *Design of Cray XD1 RapidArray Transport Core*. Mendota, MN, USA.
- [Cray Inc. 2005g] Cray Inc. (2005g). *Design of Cray XD1 RapidArray Transport Core*. Mendota Heights, MN, USA.
- [Estrin and Viswanathan 1962] Estrin, G. and Viswanathan, C. R. (1962). Organization of a “fixed-plus-variable” structure computer for computation of eigenvalues and eigenvectors of real symmetric matrices. *J. ACM*, 9(1):41–60.
- [Fernando et al. 2005] Fernando, J., Dalessandro, D., Devulapalli, A., and Wohlever, K. (2005). Accelerated FPGA based encryption. New Mexico, USA. The Cray User Group.
- [Gokhale and Graham 2005a] Gokhale, M. and Graham, P. S. (2005a). Reconfigurable computing: Accelerating computation with field-programmable gate arrays. In *Reconfigurable Computing*, New Mexico, USA. Springer.
- [Gokhale and Graham 2005b] Gokhale, M. and Graham, P. S. (2005b). *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*.
- [Shan 2006] Shan, A. (2006). Heterogeneous processing: a strategy for augmenting moore’s law. <http://www.linuxjournal.com/article/8368>. Acessado em 10/09/2009.
- [Symphony EDA 2005] Symphony EDA (2005). Symphony eda. <http://www.symphonyeda.com>. Acessado em 10/09/2009.
- [Universidade Federal de Itajubá] Universidade Federal de Itajubá. Tutorial de vhdl. Tutorial desenvolvido pelo grupo de microeletrônica.
- [Wain et al. 2006] Wain, R., Bush, I., Guest, M., Deegan, M., Kozin, I., and Kitchen, C. (2006). An overview of FPGAs and FPGA programming; initial experiences at Daresbury. pages 2–4, Daresbury, Cheshire, UK. Council for the Central Laboratory of the Research Councils.